09:53:28 OCA PAD INITIATION - PROJECT HEADER INFORMATION 02/01/89

Active

Project #: D-48-638 Cost share #: Rev #: 0
Center # : R6673-0A0 Center shr #: OCA file #: 119
Work type : RES
Contract#: DACA88-88-D-0020-0008 Mod #: Document : DO
Prime #: Contract entity: GTRC

Subprojects ? : N
Main project #:

Project unit: ARCH COLL Unit code: 02.010.164
Project director(s):
CIRCEO L JR ARCH COLL (404)894-3390

Sponsor/division names: ARMY / CON ENG RES LAB, IL
Sponsor/division codes: 102 / 020

Award period: 890118 to 900117 (performance) 900117 (reports)

Sponsor amount	New this change	Total to date
Contract value	55,043.00	55,043.00
Funded	55,043.00	55,043.00
Cost sharing amount		0.00

Does subcontracting plan apply ?: N

Title: DEVELOPMENT OF PROTOTYPE OF THE KNOWLEDGE MANAGE

PROJECT ADMINISTRATION DATA

OCA contact: William F. Brown 894-4820

Sponsor technical contact

MS. BEVERLY COSKUNOGLU
(217)373-7284
US ARMY CONSTRUCT. ENGR. RES. LAB.
NEMARK DR., P.O. BOX 4005
CHAMPAIGN, IL 61820-1305

Sponsor issuing office

MS. V. IVERSON/CONTRACTS BRANCH
(217)373-6798
US ARMY CONSTRUCT. ENGR. RES. LAB.
2902 NEWMARK DR., P.O. 4005
CHAMPAIGN, IL 61820-1305

Security class (U,C,S,TS) : U ONR resident rep. is ACO (Y/N): N
Defense priority rating : DO-C2 NA supplemental sheet
Equipment title vests with: Sponsor X GIT
NONE PROPOSED

Administrative comments -
INITIATION OF D-48-638.THIS IS A FIXED PRICE D.O. SUBJECT TO THE TERMS OF BOA DACA88-88-D-0020 (OCA FILE #119).

GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 04/24/90

Project No. D-48-638 Center No. R6673-0A0

Project Director CIRCEO L JR School/Lab ARCH COLL

Sponsor ARMY/CON ENG RES LAB, IL

Contract/Grant No. DACA88-88-D-0020-0008 Contract Entity GTRC

Prime Contract No.

Title DEVELOPMENT OF PROTOTYPE OF THE KNOWLEDGE MANAGE

Effective Completion Date 900117 (Performance) 900117 (Reports)

Closeout Actions Required:	Y/N	Date Submitted
Final Invoice or Copy of Final Invoice	Y	
Final Report of Inventions and/or Subcontracts	Y	
Government Property Inventory & Related Certificate	Y	
Classified Material Certificate	N	
Release and Assignment	Y	
Other	N	

Comments

Subproject Under Main Project No.

Continues Project No.

Distribution Required:

Project Director	Y
Administrative Network Representative	Y
GTRI Accounting/Grants and Contracts	Y
Procurement/Supply Services	Y
Research Property Managment	Y
Research Security Services	N
Reports Coordinator (OCA)	Y
GTRC	Y
Project File	Y
Other	N
	N

NOTE: Final Patent Questionnaire sent to PDPI.

Georgia Institute of Technology
A UNIT OF THE UNIVERSITY SYSTEM OF GEORGIA
ATLANTA, GEORGIA 30332

SCHOOL OF
CIVIL ENGINEERING
LELAND S. RIGGS

TELEX: 542507 GTRC OCA ATL

TELEPHONE
(404) 894-2246

March 21, 1989

Beverly Coskunoglu
USA-CERL
Facility Systems Division
P.O. Box 4005
Champaign, Illinois 61820

Ref: Knowledge Manager Project

Dear Beverly,

Enclosed is a copy of our first monthly status report previously transmitted by PAX mail. Please call George Olive or me if there are any questions.

Cordially,

Leland S. Riggs
Co-Principal Investigator

Cy: Linda Martinson
Lou Circeo
George Olive
William Brown, OCA (Ref D-48-638)

RFQ DACA88-89-Q-0143
DACA88-88-D-0020

MONTHLY STATUS REPORT

Date: March 21, 1989

Attn: Beverly Coskunoglu

Ref: Contract No. DACA88-88-D-0020, Georgia Tech Research Corporation Delivery Order # 0008

Subject: Progress Report for "Develop A Prototype of the Knowledge Manager (KM)".

I. REPORTING PERIOD: 01 Feb. - 28 Feb. 1989

II. STATUS OF ACTIVITIES:

TASK 1 -

(a) The review of the previous contracts and the existing software is complete. A report summarizing this review has been delivered.

III. NEW OR OUTSTANDING ITEMS.

None

IV. BUDGET REPORT:

Budget Report
DACA88-83-D-0020
Period: Feb 1, - Feb 28, 1989

Contract Amount
Personal Services: $54,043.00
Supplies: $1,000.00
Total: $55,043.00

VOUCHER AMOUNTS

Month Ending	Labor Task			Other Costs
	1	2	3	
Feb 28, 1989	$6,933.33	$0.00	$0.00	$0.00
Total	$6,933.33	$0.00	$0.00	$0.00

SUMMARY

Task No.	Percent Total	Total Budget	Percent Complete	Total Spent	Dollars Remaining
1	33.33%	$18,014.33	38.49%	$6,933.33	$11,081.00
2	33.33%	$18,014.33	0.00%	$0.00	$18,014.33
3	33.33%	$18,014.33	0.00%	$0.00	$18,014.33
TOTAL LABOR	100.00%	$54,043.00	12.83%	$6,933.33	$47,109.67
TOTAL OTHER		$1,000.00		$0.00	$1,000.00
D.O. TOTAL		$55,043.00		$6,933.33	$48,109.67

RFQ DACA88-89-Q-0143
DACA88-88-D-0020

MONTHLY STATUS REPORT

Date: July 15, 1989

Attn: Beverly Coskunoglu

Ref: Contract No. DACA88-88-D-0020, Georgia Tech Research Corporation Delivery Order # 0008

Subject: Progress Report for "Develop A Prototype of the Knowledge Manager (KM)".

I. REPORTING PERIOD: 01 June - 30 June 1989

II. STATUS OF ACTIVITIES:

Presented detail design

On June 16, 1989 a meeting was held at Georgia Tech to review the detail design document prepared by Georgia Tech. Present at this meeting were, Beverly Coskunoglu and Wayne Schmidt representing USA-CERL, and George Olive representing Georgia Tech.

USA-CERL suggested two major changes to the design that was presented by Georgia Tech. First, it was decided that all data files relating to subtasks should reside on each Knowledge Worker's personal computer (rather than on the PAX system as the proposed design specified). Second, changes recommended to the user interface regarding the automatic execution of a subtask.

Georgia Tech supported these modifications, and agreed to rewrite the detailed design document to incorporate them.

III. NEW OR OUTSTANDING ITEMS.

During the month of July, Georgia Tech will deliver the new detailed design document to USA-CERL. Georgia Tech will also deliver to USA-CERL a revised schedule for the implementation of the Knowledge Manager Prototype.

DACA88-88-D-0020/0008

MONTHLY STATUS REPORT

Date: July 29, 1989

Attn: Beverly Coskunoglu

Ref: Contract No. DACA88-88-D-0020, Georgia Tech Research Corporation Delivery Order # 0008

Subject: Progress Report for "Develop A Prototype of the Knowledge Manager (KM)".

I. REPORTING PERIOD: 01 Mar. - 31 Mar. 1989

II. STATUS OF ACTIVITIES:

Visited CEEC-P offices in Washington, DC.

On March 14, 1989 a meeting was held at the CEEC-P offices in Washington, DC. The purpose of this meeting was to analyze the hardware and software environment at CEEC-P as well as to formally meet the people designated to use the prototype Knowledge Manager system.

The visiting team consisting of Beverly Coskunoglu (CERL), Leland Riggs (Georgia Tech), and George Olive (Georgia Tech) spent two days at CEEC-P interviewing the Knowledge Workers. The Knowledge workers we interviewed were Jerry Goudelocke, Bill Crambo, Peggy Endicott, and Mike Rice.

The meeting began with Beverly Coskunoglu discussing the goals of the redesign effort. This was followed by a short presentation of backgrounds of the Georgia Tech team members. Following this, John Sheehey the OCE sponsor for the Knowledge Manager project elaborated on his goals for Knowledge Manager system.

Submitted report containing the details of our visit to CEEC-P.

On March 30, 1989 Georgia Tech submitted a report which gave the details of the visit to CEEC-P. This report contained the Georgia Tech analysis of the hardware and software environment at CEEC-P with particular regard to PPBES. Submission of this report fulfills the contractually required deliverable number 3.

III. NEW OR OUTSTANDING ITEMS.

During the month of April, Georgia Tech will make preliminary recommendations for the hardware and software to use in the development of the prototype. These recommendations will be submitted in writing to CERL. Following that report Leland Riggs and George Olive will attend a two day work shop at CERL to discuss these recommendations and modify them if necessary.

During our analysis to date we have discovered two points relating to scheduling that should be discussed. First, while the contract calls for a cpm type scheduler our study of the schedules indicate that this may not be necessary. Secondly, we need to agree on some basic scheduling terms, such as task, event, etc. These points should be added to the agenda for the work shop at CERL.

IV. BUDGET REPORT:

Budget Report	Contract Amount	
DACA88-88-D-0020/0008	Labor:	$54,043.00
Period: Feb 1, - Feb 28, 1989	Other:	$1,000.00
	Total:	$55,043.00

VOUCHER AMOUNTS

Month Ending	Labor Task 1	Labor Task 2	Labor Task 3	Other Costs
Feb 28, 1989	$6,693.33	$0.00	$0.00	$0.00
Mar 23, 1989				$221.00
Mar 31, 1989	$5,315.10			
Total	$12,008.43	$0.00	$0.00	$221.00

SUMMARY

Task No.	Percent Total	Total Budget	Percent Complete	Total Spent	Dollars Remaining
1	50.80%	$27,453.73	43.74%	$12,008.43	$15,445.30
2	35.45%	$19,159.97	0.00%	$0.00	$19,159.97
3	13.75%	$7,429.30	0.00%	$0.00	$7,429.30
TOTAL LABOR	100.00%	$54,043.00	22.22%	$12,008.43	$42,034.57
TOTAL OTHER		$1,000.00		$221.00	$779.00
D.O. TOTAL		$55,043.00		$12,229.43	$42,813.57

RFQ DACA88-89-Q-0143
DACA88-88-D-0020

MONTHLY STATUS REPORT

Date: July 29, 1989

Attn: Beverly Coskunoglu

Ref: Contract No. DACA88-88-D-0020, Georgia Tech Research Corporation Delivery Order # 0008

Subject: Progress Report for "Develop A Prototype of the Knowledge Manager (KM)".

I. REPORTING PERIOD: 01 Apr. - 30 Apr. 1989

II. STATUS OF ACTIVITIES:

Presented preliminary recommendations for hardware and software

At the beginning of April Georgia Tech submitted in writing to USA-CERL preliminary recommendations for the hardware and software to be used in the prototype effort. Following this, on April 13-14, 1989, Leland Riggs and George Olive meet with Beverly Coskunoglu, Wayne Schmidt, Kevin Stewart, Edward Japel, and Connie Raaymaker and USA-CERL to discuss these recommendations.

During this meeting we discussed the relative advantages of alternative hardware and software platforms. Additionally, Wayne Schmidt discussed the future directions of the computing environment in the Army.

Submitted detailed summary of Meeting.

The week after the meeting at USA-CERL, Georgia Tech submitted a detailed report summarizing the meeting, and documenting our conclusions.

The delivery of this report fulfills requirement number 4 in the contract.

III. NEW OR OUTSTANDING ITEMS.

At the workshop we had at USA-CERL, we agreed that Georgia Tech could combine the preliminary functionality report, and the specification report into one report. During the month of May, Georgia Tech will work on this report.

IV. BUDGET REPORT:

Budget Report
DACA88-88-D-0020/0008
Period: Feb 1, - Feb 28, 1989

Contract Amount

Labor:	$54,043.00
Other:	$1,000.00
Total:	$55,043.00

VOUCHER AMOUNTS

Month Ending	Labor Task 1	Labor Task 2	Labor Task 3	Other Costs
Feb 28, 1989	$6,693.33	$0.00	$0.00	$0.00
Mar 23, 1989				$221.00
Mar 31, 1989	$5,315.10			
Apr 30, 1989	$5,315.10			
Total	$17,323.53	$0.00	$0.00	$221.00

SUMMARY

Task No.	Percent Total	Total Budget	Percent Complete	Total Spent	Dollars Remaining
1	50.80%	$27,453.73	63.10%	$17,323.53	$10,130.20
2	35.45%	$19,159.97	0.00%	$0.00	$19,159.97
3	13.75%	$7,429.30	0.00%	$0.00	$7,429.30
TOTAL LABOR	100.00%	$54,043.00	32.06%	$17,323.53	$36,719.47
TOTAL OTHER		$1,000.00		$221.00	$779.00
D.O. TOTAL		$55,043.00		$17,544.53	$37,498.47

RFQ DACA88-89-Q-0143
DACA88-88-D-0020

MONTHLY STATUS REPORT

Date:

Attn: Beverly Coskunoglu

Ref: Contract No. DACA88-88-D-0020, Georgia Tech Research Corporation Delivery Order # 0008

Subject: Progress Report for "Develop A Prototype of the Knowledge Manager (KM)".

I. REPORTING PERIOD: 01 July - 31 July 1989

II. STATUS OF ACTIVITIES:

Delivered updated design document

Georgia Tech agreed to rewrite the detailed design document to reflect the changes discussed at the 16 June meeting at Georgia Tech. This document was delivered to USA-CERL on 14 July.

III. NEW OR OUTSTANDING ITEMS.

During discussions between Beverly Coskunoglu (USA-CERL) and George Olive (Georgia Tech), it was decided that the status report was the best place to note any implementations decisions. Implementation decisions would cover any decision made in coding the Knowledge Manager prototype which should be reviewed before coding the production version of the Knowledge Manager.

Georgia Institute of Technology
A UNIT OF THE UNIVERSITY SYSTEM OF GEORGIA
ATLANTA, GEORGIA 30332

SCHOOL OF
CIVIL ENGINEERING
LELAND S. RIGGS

TELEX: 542507 GTRC OCA ATL

TELEPHONE
(404) 894-2246

February 21, 1989

Beverly Coskunoglu
Facility Systems Division
USA-CERL
P.O. Box 4005
Champaign, Illinois 61820

Dear Beverly,

Enclosed is a copy of our report to you for task 1-1 under the Knowledge Manager contract. Although a copy has already been sent by PAX mail, we will continue to provide you a hard copy of the periodic reports.

Please call George Olive or me if there are any questions.

Cordially,

Leland S. Riggs
Co-Principal Investigator

Cy: Linda Martinson
Lou Circeo
George Olive
William Brown , OCA (Ref D-48-638)

KNOWLEDGE MANAGER PROTOTYPE DESIGN

Prepared Under Contract
DACA88-88-D-0020

Task 1-1

Submitted to:

Department of the Army

CONSTRUCTION ENGINEERING RESEARCH LABORATORY

Champaign, Illinois

Prepared February 21, 1989 by:
GEORGIA INSTITUTE OF TECHNOLOGY
Atlanta, Georgia

1. Overview

This report is an analysis of the current status of the effort to automate the planning and budgeting processes of the Military Construction Army (MCA) Program. The current systems were developed as the result of two contracts.

The first contract "Services to Develop a Budget Submissions Documentation Handbook for the PAX System" [DACA-88-86-C-008] was awarded to BDM Corporation of McLean, VA. This consisted of an analysis of the standing operating procedures (SOP) of the Programming and Execution Support Office, Engineering and Construction Directorate (CEEC-P) for maintaining the Army Planning, Programming, Budgeting, and Execution System (PPBES). This contract resulted in SOP Handbooks being written for two different knowledge workers (KW) at CEEC-P.

The second contract "Services to Provide a Programming Handbook on Military Resourcing & Development of Database & Retrieval Software" [DACA88-87-C-0020] was awarded to Systems Support Group (SSG) of Rockville, Md. This contract involved the same task as the first contract for a different organization, the Military Construction Branch, Assistant Chief of Engineers (ZCP-A). This resulted in an SOP handbook and software which runs on the IBM Personal Computer. The software is essentially an electronic form of the SOP handbook.

Both contractors store the daily activities of the knowledge workers in databases on the Programming, Administration and Execution (PAX) System. Additionally, the ZCP-A system also stores the activity descriptions on the PAX system. This allows for a centralized updating of the information.

2. Detailed Analysis

a. The PAX System

The PAX system is a worldwide timesharing network supporting electronic mail, access to databases, and centralized file storage. This system is maintained by the McDonnell Douglas Information Systems Group, and is accessed through TYMNET.

McDonnell Douglas provides a communications program called TymCOMM which allows a user to communicate with the PAX system using an IBM Personal Computer. This program allows the creation of script files to automate routine communications tasks such as: logging on, transferring files, and retrieving mail. This scripting feature is extremely powerful, and performs well.

The TymCOMM program also contains an error correcting file transfer function which can be invoked by the host computer. This allows a programmer to write programs on the host computer which will automatically transfer files to and from the user's personal computer. This greatly simplifies the task of moving data back and forth to a central repository.

It is not necessary to use the TymCOMM program to communicate with the PAX system, although this is certainly the easiest method. Other programs such as Crosstalk, Procomm, or any program capable of emulating a VT-100 terminal will work; however, since the file transfer capability is proprietary to McDonnell Douglas they will not be useful for transferring data files.

b. The SOP Handbooks

Both the BDM and the SSG efforts resulted in the production of printed handbooks. Each handbook contained procedures for completing an activity, and is organized one activity per page. The formats of the handbooks produced as a result of the two contracts is similar, although the SSG handbooks are organized a little better than the BDM handbooks.

Both contractors assigned each activity a number and the handbooks are organized with the activities in numerical order. In the SSG handbook the supporting documentation for an activity immediately follows the activity description page.

c. The software

The software supplied under both contacts is designed to accomplish the same thing, to generate the list of activities a knowledge worker should perform at a specific time. However, the SSG software goes considerably further in achieving that goal.

Both software systems maintain a central database on PAX. The FOCUS database software was chosen by both contractors for this purpose. Additionally, both contractors provided retrieval software to allow a knowledge worker to retrieve a "to do list" from the system. This list contains the tasks which the KW must perform in the near future as well as the activity reference number to allow the KW to look up the task in the SOP handbook.

The BDM software only prints this list on the KW's terminal, but the SSG software downloads this list to the KW's pc and provides a Microsoft Windows application to view the list. This viewer also allows the user to see the associated handbook pages on the computer rather than using the printed manual by simply clicking on the schedule line.

3. Summary and Conclusions

The approach of both contractors in solving this problem was the same, but SSG produced a more complete and useable product. However, there is still room for improvement in the SSG product.

The first problem with SSG's software is the difficulty of installing it. The software consists of a large number of batch files, communications script files, control files and data files. The user is required to set up directories place these files in them and then make modifications to some of them before the software can work. This is too complicated for the average user, and requires a programmer to do the installation. As the number of KWs increase this could cause a very large support problem.

The multitasking environment provided by SSG's choice of MS Windows is advantageous because a KW, like any office worker, has to perform many duties simultaneously. Telephone calls interrupt the task at hand, requiring access to notes or files; accomplishing some task may require the KW to run a spreadsheet or some other program. It is very useful to be able to do this without stopping the current task. MS Windows is not the only product that accomplishes this integration, however, it provides one of the most consistent user interfaces available on a IBM PC platform. This will reduce the amount of user training required for a new user.

The Windows software provided by SSG was fairly well laid out but needs some improvement on the consistency of the user menu choices. Also, it would be useful if any place a user could click the mouse on and get more information was marked somehow. It is not immediately obvious that clicking the mouse on a step will get more information.

Another problem is caused by the communications program and the knowledge base viewer not running in the same environment. The viewer runs under MS Windows, and the communications program will not. This causes the KW to constantly be switching back and forth between Windows and DOS, and the software was prone to failing at this point. The Windows environment provides a very useful interface, and ideally the communications software should run under Windows also.

The ability to produce a communication program running under windows is restricted by the lack of any non proprietary file transfer protocols on the PAX system. It should be possible to install the Kermit file transfer software on the PAX system. Another possibility would be to get McDonnell Douglas to rewrite the TymComm software. However, This would undoubtedly be expensive.

Another place to concentrate on improving is the actual downloading of the data files. Task descriptions files should be compressed to shorten the transfer time required in downloading them. It appears that these files are transferred every time a connection is made, whether or not the version of them on the pc is out of date. No information that is not out of date should ever be transferred.

Another improvement in performance would be achieved if the database used for scheduling was more directly accessible by the KW's pc. This could be achieved by storing it on a network server, and setting up a program on the server to make sure the local copy of the database agreed with the one on PAX. This would eliminate the need for the file transfer software on the KW's machine as well as speed up access to the schedules.

4. Recommendations

After reviewing the results of the previous efforts, the following improvements are recommended.

a. Simplify the software installation procedure.

b. Modify the Windows software to give it a more uniform user interface.

c. Investigate the addition of communications protocols other than the Tymcomm protocol.

d. Compress the data files where possible.

e. Change the communications system so that it will only download files which are out of date on the pc.

f. Split the data files into two groups, those that change often and those that do not.

g. Investigate the use of a local area network with a central server communicating with PAX.

Georgia Institute of Technology
A UNIT OF THE UNIVERSITY SYSTEM OF GEORGIA
ATLANTA, GEORGIA 30332

SCHOOL OF
CIVIL ENGINEERING
LELAND S. RIGGS

TELEX. 542507 GTRC OCA ATL

TELEPHONE
(404) 894-2246

March 30, 1989

Beverly Coskunoglu
Facility Systems Division
USA-CERL
P.O. Box 4005
Champaign, Illinois 61820

Dear Beverly,

Enclosed is a copy of our report to you for task 1-2 under the Knowledge Manager contract.

As we indicated in our last report, we will continue to provide you a hard copy of the periodic reports even though we have previously sent a copy by PAX mail.

Please call George Olive or me if there are any questions.

Cordially,

Leland S. Riggs
Co-Principal Investigator

Cy: Linda Martinson
Lou Circeo
George Olive
William Brown , OCA (Ref D-48-638)

KNOWLEDGE MANAGER PROTOTYPE DESIGN

Prepared Under Contract
DACA88-88-D-0020/0008

Task 1-2

Submitted to:

Department of the Army
CONSTRUCTION ENGINEERING RESEARCH LABORATORY
Champaign, Illinois

Prepared March 30, 1989 by:
GEORGIA INSTITUTE OF TECHNOLOGY
Atlanta, Georgia

1. Overview

This report contains our analysis of the hardware and software environment at the Programming and Execution Support Office, Engineering and Construction Directorate (CEEC-P). Action officers, or Knowledge Workers (KW), in this office are involved with Military Construction, Army (MCA) aspects of the Army Planning, Programming, Budgeting, and Execution System (PPBES).

The particular workers designated to use the Knowledge Manager (KM) prototype oversee portions of the Construction Appropriations Programming Control and Execution System (CAPCES), and the DD Form 1391 Processor. In the course of their duties, there are a number of deadlines for reports they must produce. These deadlines may be mandated by the Office of Management and Budget (OMB), Office of the Secretary of Defense (OSD), the Office of the Secretary of the Army (OSA), or the Office of the Chief of Engineers (OCE).

Each worker, in preparation for one of these major events, must perform a number of tasks. The deadline each task is set by the worker with the constraint that all the tasks must be finished by the mandated deadline.

2. Hardware

Each CEEC-P Knowledge Worker will have a Compaq Deskpro 386/e personal computer connected to a 3Com network. Each computer will be equipped with a 110 megabyte hard disk, 4 megabytes of main memory, a VGA monitor, a 40 megabyte tape backup unit, and a math coprocessor. Additionally, each KW remains constantly connected to the PAX system through a communications server also connected to the network.

3. Software

On the personal computers, the only software the CEEC-P KWs are consistently using is Word Perfect and TymCOMM. CEEC-P knowledge workers are currently using version 4.2 of WordPerfect, however version 5.0 is available through the network. In addition, John Sheehey is purchasing Microsoft Windows 386 for each KW workstation in his branch.

There are several other programs being considered for use on the personal computers. These are OS/2, PC-FOCUS, and Vistacomm. OS/2 is being considered because it provides a better multitasking environment. PC-FOCUS is being considered because it allows users to access the PAX data from a menu driven interface running on the personal computer. Finally, a communication package called VistaCom is currently being considered because VistaCom provides a programmer's toolkit which allows greater control of the file transfer process than that offered through TymComm. The Corps of Engineers has a site license for VistaCom.

The data the KWs use is accessed through CAPCES and the DD Form 1391 Processor system, two PAX-based applications. It is contained in FOCUS databases. FOCUS programs called FOCEXECs exist to scan these databases for changes made by the individual installations, as well as to generate the reports the KWs need to perform their job. John Sheehey has specified that all the information contained in the Knowledge Manager system must be accessible to the KWs through the PAX system.

Pc-dugout, a program developed by CERL, automates the updating of data files on the personal computer, from the PAX system. This program runs in conjunction with the TymComm program. It allows the PAX system to maintain a current set of files for any application, and insure that each personal computer has the most current version of every file in the set.

There is one other system in use at CEEC-P that is not included in this study but deserves mentioning. This is a building wide electronic mail and meeting scheduling system called OASIS. An extension of the scheduler in the Knowledge Manager which would access OASIS and adjust deadlines based on time blocked out by meetings scheduled on the OASIS system might be useful at some future date.

4. Conclusion

Some CEEC-P Knowledge Workers are not running any applications on their personal computers except one, TymComm. They are only using their computer as terminals to access the PAX system. If the Knowledge Manager system is to succeed in gaining acceptance by these users it must be easy to use, provide an enticing interface, as well as perform the scheduling task well.

For the Knowledge Manager system to be successful, it is critical that it provide all the functionality of the current systems as well. This means accessing programs such as Word Perfect in addition to providing a seamless interface to data produced by PAX-based applications.

KNOWLEDGE MANAGER PROTOTYPE DESIGN

Prepared Under Contract
DACA88-88-D-0020/0008

Task 1-3

Submitted to:

Department of the Army
CONSTRUCTION ENGINEERING RESEARCH LABORATORY
Champaign, Illinois

Prepared April 12, 1989 by:
GEORGIA INSTITUTE OF TECHNOLOGY
Atlanta, Georgia

PRELIMINARY RECOMMENDATIONS OF HARDWARE AND SOFTWARE

1. Introduction

This report summarizes our analysis of the environment in which the prototype Knowledge Manager system will be used. Our preliminary recommendations for the hardware and software necessary to develop the prototype are discussed below. These recommendations are based upon our review of previous efforts to automate the Knowledge Worker's tasks and interviews with the CEEC-P Knowledge Workers.

2. Recommended Software

The major emphasis of the prototype Knowledge Manager system is the scheduling component. Our interviews with the CEEC-P Knowledge Workers revealed that their schedules consist of a set of loosely related goals which are accomplished by performing a group of sequential tasks. A commercial software package is not needed to achieve the user's scheduling requirements; this functionality can be better accomplished by developing our own scheduling software. In addition by writing our own scheduler we can customize the means by which users interact with the software.

We recommend writing the prototype Knowledge Manager as a Microsoft Windows application. The language we recommend using is the Actor programming language published by the Whitewater Group. We chose this approach because the Knowledge Manager system must be easy to learn, easy to use, and compatible with existing software. Additionally, because of the evolutionary process of developing a prototype, preference was given to software which would allow for rapid implementation of changes to the prototype design. Once the prototype is completed these selections can be readily changed for implementation of the production version of the Knowledge Manager system.

Systems that provide a graphical user interface have been shown to be easy to learn, and use. Some examples of standard graphical user interface systems currently on the market are NeWS, X Windows, the Macintosh interface, Microsoft Windows, and Microsoft Presentation Manager. All of these systems draw heavily from the same source, the Smalltalk interface.

Smalltalk was developed at the Xerox Palo Alto Research Center (PARC). It resulted from a project with the goal of designing a computer system that was easy to use. The main features of this system were a graphical interface on which the data and all possible actions the user could perform on that data are displayed on the screen. The user interacted with this system by selecting the desired actions with a mouse. The programming language of this system was Smalltalk.

The Smalltalk language is an object-oriented language. This means that code and data for a routine are combined in something called an object. Other objects can not access this code or data directly; instead, they send messages to an object telling it to perform some action on itself and return the results. This is, in a sense, the ultimate in structured programming. No other object in the system requires any knowledge about how any other object performs its function; rather it only needs to know what function the other objects perform.The advantages of this type of development environment are rapid development and maintainability.

Rapid development is inherent in an object-oriented system because of the concept of reusability. Reusability means if there is some object that performs most of some task, the programmer may produce a new object which is a descendant of this object, and add only code for actions which will be performed differently. Also, because the programming tools are built into the development environment, it is easy to make a small change to a program and see the effects of this change quickly. Furthermore, when a program fails, this type of system will provide a complete execution history to the point of failure, and even allow the error to be corrected and continue execution as if the error had not occurred.

Maintainability is provided by the structured nature of this type of development. Once the maintainer understands the function of the objects in the system, changes to the system or problem fixes are localized to the object or objects which have responsibility for the part of the system in question. The programming language Actor, like Smalltalk, provides the advantages of object-oriented programming with the added advantage of running under Microsoft Windows. Actor provides a complete programming environment for the rapid development of MS Windows applications.

The advantage of using MS Windows is, its compatibility with IBM's System Application Architecture Common User Interface standard. This standard is IBM's declared interface for the 1990s. By adhering to this standard it will become more likely that programs will be able to provide a uniform interface to the user across many hardware platforms from IBM mainframes down to personal computers.

This approach, while recommended, is not the only approach. There are several other object-oriented languages on the market, or soon to be on the market. Several of these systems based on C++ will be on the market soon. Since most of these are beta releases, we do not feel that working with them would be wise in the production of the prototype. However, because such systems would produce smaller programs they may be good candidates for the production version of the Knowledge Manager.

3. Recommended Hardware

The hardware already in place at CEEC-P, Compaq deskpro 386/e microcomputers, are compatible with the largest base of personal computers in business use. Also, they are more than capable of running the recommended software. Therefore, we recommend using these systems.

In addition to the equipment they already have, each user will need a mouse attached to their machine. Even though it is possible to run a Microsoft Windows application without a mouse, it is extremely frustrating for the user and will prevent them from using the Knowledge Manager effectively.

We recommend an additional AT type personal computer be dedicated to communications between CEEC-P network-linked Knowledge Workers and PAX. The role of this PC would be to maintain synchronization between Knowledge Manager files on the network server and those on the mainframe. This would be an advantage because it would allow these Knowledge Workers to use more of their personal computer resources to perform their job rather than wasting time doing "bookkeeping" tasks for the Knowledge Manager. This approach would not prevent users from accessing PAX-based applications whenever they chose.

4. Summary

We recommend the following:

- Develop customized scheduling software.
- Use Microsoft Windows/386 as the primary system software.
- Use the Actor programming language as the development software.
- Use the existing CEEC-P computer hardware.
- Add a mouse to each Knowledge Worker's personal computer.
- And, dedicate an additional PC to act as a communications server between CEEC-P Knowledge Workers the PAX system.

KNOWLEDGE MANAGER PROTOTYPE DESIGN

Prepared Under Contract
DACA88-88-D-0020/0008

Task 1-5
Detailed Design Of Knowledge Manager Prototype

Submitted to:

Department of the Army
CONSTRUCTION ENGINEERING RESEARCH LABORATORY
Champaign, Illinois

Prepared June 12, 1989 by:
GEORGIA INSTITUTE OF TECHNOLOGY
Atlanta, Georgia

Table Of Contents

Goals

The goal of the of the Knowledge Manager (KM) system is to preserve institutional knowledge and assist action officers in the performance of their daily tasks. KM must be able to provide answers about what tasks to perform, as well as how and when to them. The system must be able to maintain and display a dynamic schedule of tasks to be performed. KM purports to free the knowledge worker (KW) from repetitive tasks by automating as many steps as possible.

The goal of this prototype to provide a useful subset of these KM functions, as well as to provide a platform on which to study the best methods for implementing the production version of the Knowledge Manager.

Overview

Two sections make up this report. The first section provides a general overview of the system. In this section the fine points of the system are only briefly described. In the second section, the interaction of the components of the Knowledge Manager is described in greater detail.

The design of the prototype Knowledge Manager is based on the object-oriented language, Actor. The program is designed to run under Microsoft Windows.

In keeping with the object-oriented design of the Knowledge Manager, the interaction between KM components will be described in terms of the messages to which each component can respond. This general view of the system allows most of the effort in the prototype to be focused on the design of the system as it would perform on an ideal platform.

This approach will minimize the effort in implementing the production version of the Knowledge Manager. Since the system is described in terms of its reaction to messages, only the lowest level of the message-processing functions (those that deal with the operating system) will need to be modified for a different operating system.

From a general standpoint, the interaction of an object, the windows in which it is displayed, and the file that holds its data is transparent to the user. This window-object-data synthesis provides the user with the illusion that the data can be changed directly and in real-time.

For example, when a user changes the percent-complete value in one window, it appears to the user that the value of interest was selected and a new value inserted. What really happens is this: The user changes the data on the screen, the window sends a message telling the object that it has been changed, and the object performs some operating system dependent operation (discussed in detail in the Objects section) and the data is stored in a file.

The User Interface

In order to provide the user with consistent information about the status of the system, several basic themes will be followed throughout the user interface.

The use of colors

Standard colors will be used to provide the user with instant information about the status of information on the screen. So that a KW sees at a glance the status of all the subtasks on the ToDo list, the colors red, green, and yellow will be employed consistently. In particular, red will be used to indicate a situation that requires immediate attention, for instance, subtasks which are late. Yellow will be a warning color and used to draw the user's attention to some important - but not yet critical - situation. Green will indicate that an item is within normal operating constraints. The normal background color will be white; normal text will be black; and places where the user may double click the mouse and link to other windows will be light blue.

These colors will be modifiable, however, this cability will not be given to the user; rather, it will allow us to experiment with colors during prototype evaluation. The user will not be allowed to customize the colors. Color standardization is necessary to prevent a KW from accidentally misreading a situation if evaluating from a workstation where another user implemented a different color scheme.

Window types

There are five basic types of windows the user will see. These are the: 1) ToDo list, 2) Schedule window, 3) Task window, 4) Subtask window, and 5) Attachment window. Each window has slightly different properties that enable it to present its particular kind of information.

ToDo list window

The ToDo list window displays the knowledge worker's daily list of tasks (actually subtasks). This window can process hyperlinks, has knowledge of dates, and allows the KW to edit private (user-discretionary) tasks. This window is the point where a KW will initially access the KM system. It is attached to all other windows in the KW's knowledge base. The ToDo window may always be brought to the front of the on-screen windows by pressing the right-hand mouse button. The menu bar of the ToDo window contains a "News" button. When the KW selects this button, KM traverses a link to an attachment containing the daily system news.

The ToDo window contains one line for each task the KW needs to perform. A more detailed description of this task may be found by double clicking the mouse while pointing to this line; this action will bring up a subtask window.

Schedule window

The schedule window is very similar to the ToDo list window. This window will generally be the entry point to the Knowledge Manager system for users at the supervisory level and above.

This window allows the user to manipulate the schedule and gain access to the task-level information. This window is described in greater detail in the schedule section.

Task window

The task window will look identical to the subtask window (see below). The only difference between the two is their level of decomposition. A task window decomposes into subtasks; a subtask window decomposes into steps.

Subtask window

The subtask window displays due dates, the steps that make up the subtask, and associated information. This window can process hyperlinks, is editable (that is, the KW can modify the subtask description), can create/delete attachments, and can link to or execute attachments. From this window the user can link to the task-level description of the subtask and display this description in a task window.

The menu bar of a subtask window contains two choices of particular interest: "DoIt" and "Attachments". The "DoIt" menu choice causes the Knowledge Manager to automatically perform the selected step. If the currently-selected step can not be executed, the "DoIt" menu button will be grayed to indicate this fact. This automatic execution capability will be simulated in the prototype. The "Attachments" menu button will bring up a submenu which contains a list of items attached to the step currently selected.

Usually each step will have at least one item attached to it, i.e., a more detailed description of that step. If the user double clicks the mouse on a step, this link will automatically be traversed, and the associated description will be shown in either attachment window or another subtask window, depending on the type of link. Hyperlinks in a subtask window may only be attached to steps. By using the "Attachments" submenu, a large number of links can be attached to each step.

Attachments

Attachment windows are nothing more than ASCII file listers. These windows understand only one command: close. This window type represents the terminal type of the KM system; no hyperlinks may originate from one of these window types.

Hyperlinks

To give the Knowledge Manager the ability to make intelligent decisions about what action to perform when a user activates a hyperlink, the type of data contained in the attached file is also associated with the link. For instance, a hyperlink would contain information about whether the attachment contained a document or a program. In the prototype, the only types of hyperlinks the KM will be able to process are subtask links, attachment links, and execution links since automatic execution will only be simulated. When the user commands the KM to traverse a link using one of the methods described previously, the KM determines link type, creates an appropriate window type, and loads it with the data attached to the link. In the case of an executable link, the KM would spawn the designated process and pass it the attached data file.

Organization Of Data

This section describes the flow of the data through the Knowledge Manager system from the master files on the Programming, Administration, and execution system (PAX) down to the user files on each knowledge worker's PC. For purposes of clarity, some detail about the actual file formats is not included here. The complete description of the format for each file can be found in Appendix I.

The data as it resides on each KW's personal computer is in final form. That is, the scheduler has been run on it and all hypertext links are established.

ToDo list files

On each PC there will be one file containing the ToDo list. This file is the base of the Knowledge Manager system. Starting from this file, any other file can be reached by traversing the links. Basically, this file contains a list of each subtask assigned to the KW and its due date. The ToDo file is produced by combining the output of the scheduler with the KW's private schedule. Details of how to perform a subtask are found by traversing a link associated with a subtask to the associated description.

In addition, a knowledge worker can add any number of user-discretionary subtasks to the ToDo list. These subtasks are considered private to the individual knowledge worker and not moved back to the central archive. All files associated with private subtasks will be stored under a separate directory on the user's hard disk. This storage strategy will allow the user to find the files easily for backup purposes, and prevent naming conflicts with Knowledge Manager subtasks.

Schedule files

No files will be stored exclusively for the schedule. The data used by the schedule window will be extracted from the task files when the schedule window is opened. The small number of tasks allowed in the prototype will prevent this operation from taking a long time. However, there will be a longer delay when opening a schedule window than when opening a ToDo window.

In the production version of the Knowledge Manager, the joining of the data from all the task objects should be done by a database server.

Task files

The task description file is interesting in that it is viewed differently depending upon the level in the system from which it is accessed. At the system level, the task description file is the fundamental entity for the scheduler. From the knowledge worker's point of view, this file is merely an attachment which gives some general information about a subtask. From a supervisor's point of view, it is the entry point of the system and ties assignments together.

Subtask files

There is one file for each subtask on the ToDo list. This file contains the general description of the subtask as well as its due date, a list of steps involved in performing this subtask, and links to any attachments associated with the subtask.

Each subtask file includes a link to its task description file. This file contains general information about the task of which this subtask is part. System wide, only one task file exists for each task; each system-wide file is linked to all of its subtask files.

Attachment files

An attachment is a file of any format that contains some information (text, documents, programs, data) which might be helpful to the knowledge worker. Schedule, Task, or Subtask files can have attachments. Part of an attachment link field contains information about how to operate on that information. In the prototype, only ASCII text information can be manipulated, however, any other attached information will be carried through the system.

Hypertext links

The hypertext links are implemented by using DOS as a database system. Each file contains one record of data, with the file name being the key to the record. A hypertext link can be established by adding the name of one file to another file. This very general database system allows the hypertext links to be maintained across the diverse operating environments present within the Knowledge Manager system. In the production version, this system could be easily implemented to run under a database server.

Scheduler

Overview

Analysis of of the nature of Planning, Programming, Budgeting, and Execution System (PPBES) tasks determined that a classical critical path method (CPM) scheduler is not necessary in the Knowledge Manager system. Specifically, knowledge workers' tasks are related to each other through externally-fixed ending dates. In typical CPM schedulers, ending dates are calculated by arranging tasks to complete a process as quickly as possible. In contrast, in PPBES it is possible for all tasks to have slack time. This structure seems to preclude CPM.

Instead of specifying the type of scheduling algorithms to use, the prototype concentrated on the scheduling subsystem functionality. In its daily operation the Knowledge Manager scheduler will automatically provide supervisory level personnel with a list of tasks that should be in progress. The supervisors must be able to assign the subtasks to the knowledge workers under their supervision. The prototype must provide a mechanism to accomplish this as well as the ability to redistribute subtasks among KWs. Additionally, the prototype should keep track of a knowledge worker's schedule and block out times when a KW will be unavailable to work on a subtask.

The scheduler window is the first window that any supervisor will see. This window will display all the tasks in the knowledge base in chronological order. From this window a user can add or delete tasks, bring up a task description window, change task due dates, and do what-if analysis on the schedule.

Changing the schedule

The ability to do "what-if" analysis on the schedule will not be restricted; any user can experiment with changing task due dates. However, only authorized users can apply these changes to the master schedule. A user may restore the schedule from the master schedule at any time during a what-if analysis session.

When the user changes a due date for a task, the scheduler will run through the predecessor list and adjust related dates. If the scheduler finds any task that does not have enough working days for completion, it will be flagged as late.

When a user with sufficient authority chooses to apply the date changes to the master schedule, the scheduler broadcasts a message to each KM user's machine. Users who are currently on-line will receive this message that indicates what changes have been made to the schedule. Other users will receive a schedule-update message upon subsequent login to KM.

Adding or deleting tasks

Only authorized users can add or delete a task. If a user does not have this privilege, menu buttons will be grayed to indicate that this option is not available.

When a task is added or deleted, a taskAdded or taskDeleted message is sent to users currently logged into KM. Other users will receive a similar message upon subsequent login to the system.

Accessing task window

When a user double clicks on a task line in the scheduler window, the associated task window is opened on the screen.

Messages

The passing of messages between objects is primarily performed by Actor. Some Actor messages are carried by the Windows message queue. In either case no special programming needs to be done to implement the messaging system. However, system-wide messages of any type, as well as broadcast messages, require some special programming.

Broadcast messages

Broadcast messages will be implemented in Actor by making one class of objects the ancestor of all Knowledge Manager objects. This class will respond to all broadcast messages. Objects in this class will contain special knowledge about the system so that they can rebroadcast the messages to objects in other classes that need to have this information.

System wide messages

System-wide messages will also be handled by the class described above. However, the loosely-coupled nature of the users of the Knowledge Manager system requires that these messages be passed in files. These files will be uploaded to PAX and then downloaded to all users.

This special programming may not be necessary in future implementations, but in the prototoype they insulate the other KM classes from any special knowledge of the system interrelations.

Detailed Object Descriptions

Task

Description

A task object accepts messages from the user and performs the specified actions on a task description file. If any message should cause the contents of the corresponding task description file to change, the file will immediately be written to disk to protect it in case of system failure.

A subtask object is created when a user selects an item on the ToDo list. The file specified by the hyperlink will be associated with it at that time.

There may be multiple subtasks for each task, and they may be assigned to different KWs.

The data file associated with this object is an ASCII file with one field on each line. These fields are in human readable form. This allows a knowledgeable user to change these files with a text editor, and provides flexibility which should prove useful in rapidly evaluating changes to the prototype.

Attachments to a task are global rather than associated with any subtask.

In the production version the file structures can, and probably will, be changed with relatively little impact on the Knowledge Manager code. Only a few of the lowest level message handlers will have to be recoded to handle the new file formats.

Fields

title

A short title for this task.

Description

A more detailed description of the task.

status

The current status of this task.

date_due

The date this subtask is due.

days_needed

Number of days needed to accomplish this task.

pct_complete

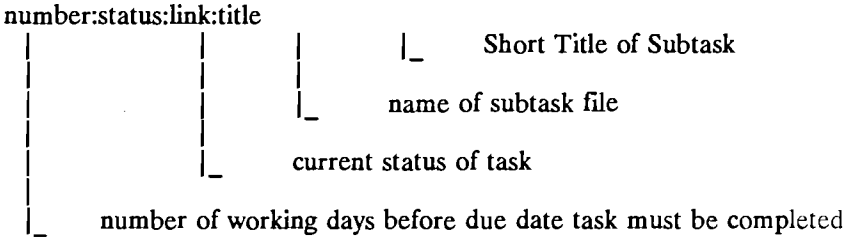
Percent task is complete

Attachments

A list of file names, one per line.

Subtask

A list of subtasks associated with this task. There is one subtask per line and the format of the line is:



dependencies

A list of the task ids of tasks which must be completed before this one can begin.

Class Messages

new(self) SUPERVISOR

An empty task with one subtask will be created.

open(self,name) ANY

A task is created and the named file is read into it and displayed.

Instance Messages

selectSubtask(self) ANY

The subtask the user is pointing to becomes the current subtask.

addSubtaskAbove(self,title) SUPERVISOR
addSubtaskBelow(self,title)

A subtask with the specified title is added above or below the currently-selected subtask.

changeSubTask(self,title) KWONLY

The title of the currently selected subtask is changed to the title specified. (The local copy of the title is changed, and a message is sent to the subtask to change its title.)

deleteSubtask(self) SUPERVISOR

A message is sent to the currently-selected subtask to delete itself.

selectAttachment(self) ANY

The attachment the user is pointing to becomes the current attachment.

addAttachment(self,type,filename,title) SUPERVISOR

The attachment file is attached globally to this task. It is given the title specified, and a hyperlink of the type specified will be used to link it to this task.

deleteAttachment(self) SUPERVISOR

The currently selected-attachment is deleted.

Subtask

Description

A subtask object accepts messages from the user and performs the specified actions on a subtask description file. If any message should cause the contents of the corresponding subtask description file to change, the file will immediately be written to disk to protect it in case of system failure.

A subtask object is created when a user selects an item on the ToDo list. The file specified by the hyperlink will be associated with it at that time.

There may be multiple subtasks for each task, and they may be assigned to different KWs.

The data file associated with this object is an ASCII file with one field on each line. These fields are in human readable form. This allows a knowledgeable user to change these files with a text editor, and provides flexibility which should prove useful in rapidly evaluating changes to the prototype.

In the production version the file structures can, and probably will, be changed with relatively little impact on the Knowledge Manager code. Only a few of the lowest level message handlers will have to be recoded to handle the new file formats.

Fields

assigned_to

The id of the KW that this subtask is assigned to. This field is only editable by a supervisor.

title

A short title for this subtask.

status

The current status of this task.

date_due

The date this subtask is due.

days_before_task

The number of working days before the task is due that this subtask must be completed.

days_needed

Number of days needed to accomplish this task.

pct_complete

Percent task is complete

Attachments

A list of file names, one per line.

Steps

A list of steps. The format for a step list is:

```
Step Title
<tab> type link 1 : link 1 : Title link 1
<tab> type link 2 : link 2 : Title link 2
:
<tab> type link n : link n : Title link n
Step 2 title
:
```

Class Messages

new(self) SUPERVISOR

An empty subtask with one step will be created. The id of the person who creates it will be filled into the assigned_to field. This allows that person to continue to fill it out with the KWONLY privilege until it is assigned to some KW.

open(self,name) ANY

A subtask is created and the named file is read into it and displayed.

Instance Messages

selectStep(self) ANY

The step the user is pointing to becomes the current step.

addStepAbove(self,title) KWONLY
addStepBelow(self,title)

A step with the specified title is added above or below the currently-selected step.

changeStep(self,title) KWONLY

The title of the currently-selected step is changed to the title specified.

deleteStep(self) KWONLY

The currently-selected step is deleted if it has no attachments. If the step has attachments they must be deleted with deleteAttachment first.

selectAttachment(self) ANY

The attachment the user is pointing to becomes the current attachment.

addAttachment(self,type,filename,title) KWONLY

The attachment file is attached to the currently selected step. It is given the title specified, and a hyperlink of the type specified will be used to link it to this task.

deleteAttachment(self) KWONLY

The currently-selected attachment is deleted.

ToDo List

Description

There is only one instance of the ToDo list object. It is created when the Knowledge Manger is started, and handles the user messages relating to the subtasks the user must perform.

The ToDo List file contains on line for every task assigned to a knowledge worker. That line consists of the fields separated by comas. The only required information in this file is the subtask link, the rest of the fields are duplicated in this file to reduce file I/O.

The ToDo list for supervisors links to tasks rather than subtasks. This is how supervisors get to the tasks for assigning subtasks, and analyzing the schedule.

Fields

duedate

The data the subtask is due. The file is sorted on this field

status

The current status of this subtask. This is the secondary key for this file.

subtask

The link to the subtask file.

title

The title of this subtask.

pct_complete

The percent complete for this subtask.

Class Messages

new(self)

Issued once by system during start up. Creates a ToDo list.

Instance Messages

close(self) ANY

Closes all other Knowledge Manager windows, and exits Knowledge Manager.

selectItem(self) ANY

Sends a open(SubTask,subtask) message using the subtask field of the line pointed to by the user.

addItem(self,duedate,filename) KWONLY

Adds the item to the ToDo list according to the sorted order. Creates a subtask by sending a new message to the subtask.

deleteSelectedItem(self) KWONLY

The currently-selected item is removed from ToDo list, and a destroy message is sent to the item itself.

Attachment

Description

An attachment is a file containing any type of data. The only type of data the prototype will be able to display is, ASCII text. It is acceptable to attach any other type of data to an object, the Knowledge Manager will keep track of such files.

Fields

NONE

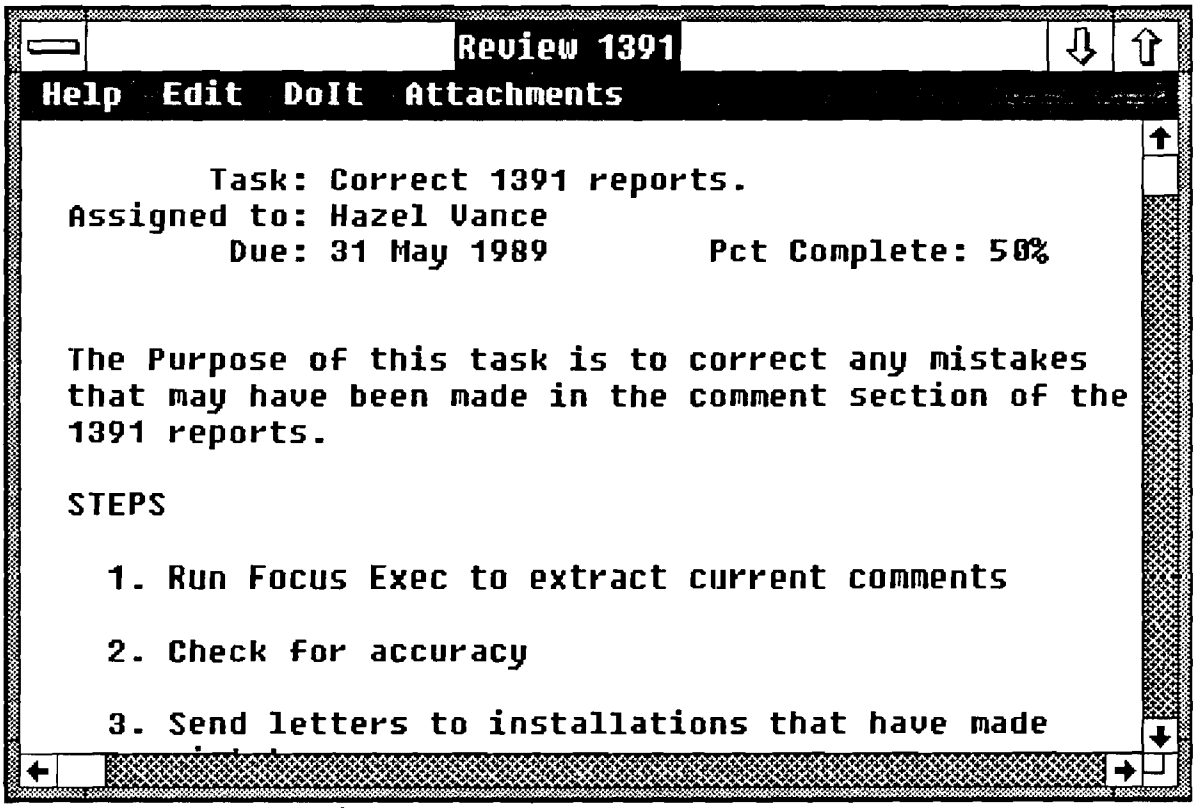
Operations

new(self,filename) ANY

Display the file in a window.

close(self) ANY

Close file and remove window.



Example Subtask Window

ToDo List Window

The ToDo list window is the first window the users sees when the system is started. Each line of the ToDo list contains one subtask. The user can select this task by pointing to it with the mouse and clicking once. If the user points to a line and double clicks the mouse, the associated link is activated and the subtask window appears.

Menu Items

Help

Invoke the Knowledge Manager help function.

Edit

This choice is only available when a ToDo list item is selected. When this choice is selected a submenu will appear with the following choices.

Add

Add a new item to the personal schedule

Delete

Delete the selected item from the personal schedule.

News

Open the news file as an attachment.

Status

Set the reporting status of an item on the list of things to do.

Complete

Mark the item completed, and remove from list of things to do.

Started

Mark the item started.

Schedule Window

The schedule window is the entry point to the Knowledge Manager system for users wishing to edit task information. The task due date is directly modifiable from this schedule window and all other task information is editable in the task window, which the schedule window will allow a user to open.

The user can select a task by pointing to it with the mouse and clicking once. If the user points to a task and double clicks the mouse, the associated link is activated and a task window appears.

Menu Items

Help

Invoke the Knowledge Manager help function.

Edit

Add

Add a new task.

Delete

Delete the currently-selected task.

Modify

Modify the currently-selected task. This is the same function invoked when the user double clicks on a task.

Schedule

WhatIf

Allows the user to change the date field on the currently-selected task. The scheduler will flag all impacted tasks.

Change

Accept the current changes into the master schedule. This button will be grayed on for uses not authorized to perform this action.

Task Window

The task window allows knowledge workers to view information about a task in general, it also allows users at the supervisor level or above access to change the task information.

Menu Items

Help

Invoke the Knowledge Manager help function.

Edit

Add

Add a new subtask to the task.

Delete

Delete the selected subtask from the task.

Modify

Open a subtask window to allow the selected subtask to be modified. This is the same as double clicking on the selected subtask.

Attachments

This is a dynamic menu. It contains a list of every attachment connected to the task. At the task level attachments are connect to the task in general not an individual subtask.

To access the attachments for a subtask the user should select modify, or double click on the subtask which will open the subtask window.

Assign

Allows the user to edit the assigned to field of the selected subtask.

Subtask Window

The task window allows knowledge workers to view information about a task in general, it also allows users at the supervisor level or above access to change the task information.

Menu Items

Help

Invoke the Knowledge Manager help function.

Edit

Add

Add a new step to the subtask.

Delete

Delete the selected step from the subtask.

Modify

Edit the wording of the selected step.

DoIt

Automatically complete the selected step. The operation of this feature will only be simulated in the prototype version of the KM.

This item will be grayed if the Knowledge Manager can not automatically complete the currently-selected step.

Attachments

This is a dynamic menu. It contains a list of every attachment connected to the selected step

Status

Set the reporting status of an item on the list of things to do.

Complete

Mark the item completed and remove from list of things to do.

Started

Mark the item started.

Attachment Window

This windows is a text file viewer. It allows the user to view an ASCII file attached to any other type window.

Menu Items

Help

Invoke the Knowledge Manager help function.

KNOWLEDGE WORKER SYSTEM PROTOTYPE

USER'S GUIDE

Oct 89

Table of Contents:

I. INTRODUCTION

What is KWS Prototype?

The Knowledge Worker System (KWS) is being developed in order to meet the following goals:

- Preserve institutional knowledge
- Provide a reminder about what, when, and how to perform tasks
- Offer assistance in executing repetitive tasks

The purpose of the KWS Prototype is to provide a useful subset of functions that should be included in the Knowledge Worker System. This prototype is intended to be a platform on which to investigate the best methods for implementing the production version of the KWS.

The KWS Prototype is not a mock-up of the production KWS. As you use the Prototype, please evaluate what functionality is helpful to you in performing your duties and consider what should be added or changed in order to make this system useful to you.

The KWS Prototype was developed using Microsoft Windows and therefore will look similar to other Windows applications. If you know how to use any other application that uses a standard Windows interface, you will find that the KWS Prototype has the same look and feel. Even if you've never used a Windows application, we hope you will find the KWS Prototype easy to use.

How to Use This Guide

This User's Guide has been developed as a reference tool for the KWS Prototype. The guide includes three main sections plus three appendices.

Section I - WINDOWS OVERVIEW

This section describes the generic Microsoft Windows structure and functionality. If you are already familiar with the Windows interface, skip this section entirely.

Section II - HOW KWS WORKS

This section describes how to navigate the KWS Prototype. That is, here is where to find information about each level of KWS and how to use each level.

Section III - HOW TO ADD/MODIFY/DELETE INFORMATION

This section documents how to use KWS to add or change information. A description of how to perform the basic KWS Prototype operations is included here.

Appendix A - HELP

This appendix includes the HELP information for each type of screen found in the Prototype. This information is taken from the Prototype HELP screens.

Appendix B - UTILITIES

This appendix lists the Microsoft Windows utilities that are included with the KWS Prototype.

Appendix C - GLOSSARY

This appendix defines several terms that the Prototype user will encounter while using the software.

II. MICROSOFT WINDOWS OVERVIEW

Background

MicroSoft Windows is an extension of the DOS operating system. Windows is a graphical system; it uses small symbols (icons) to visually convey information about the system in a condensed format. Each window is formatted with several standard components that make common operations swifter and easier than typing a command line from a DOS prompt. The standard components of a window are initiated by choosing them with the mouse and clicking on them. [**Always use the left-hand mouse button.**]

Standard Windows Components

Standard Windows components include:

System Menu Box & System Menu
The system menu box and system menu is located in the upper-left corner. A DOUBLE CLICK ON THE MENU BOX WILL CAUSE THE WINDOW TO CLOSE.

Title Bar
The title bar contains the name of the window and is located at the top in the center. CLICKING ON THE TITLE BAR AND DRAGGING IT WILL PULL THE ENTIRE WINDOW WITH IT.

Thick Border
The thick border surrounds the window. CLICKING ON THE BORDER AND DRAGGING IT WITH THE MOUSE BUTTON DEPRESSED WILL CHANGE THE WINDOW SIZE.

Minimize Box
The minimize box is located in the upper-right corner (the down-arrow icon). CLICKING ON THE MINIMIZE BOX WILL CONVERT THE ENTIRE WINDOW INTO AN ICON AND "STORE" IT IN THE LOWER-LEFT CORNER OF THE MONITOR SCREEN. THE WINDOW CAN BE EASILY RETRIEVED BY A DOUBLE-CLICK OF THE MOUSE.

Maximize Box
The maximize box is also located in the upper-right corner (the up-arrow icon). CLICKING ON THE MAXIMIZE BOX WILL CAUSE THE WINDOW TO ZOOM TO FULL SCREEN SIZE.

Restore Box
The restore box replaces the maximize box when the window is at full screen size (the up-down arrow). CLICKING ON THE RESTORE BOX WILL RESTORE THE WINDOW TO THE SIZE AND SHAPE IT HAD BEFORE ZOOMING.

Menu Bar

The menu bar is located under the title bar. This bar is used for making menu and drop-down selections. The functions in this bar are described in Section II - How KWS Works and in Appendix A - Help.

Selection Bar

The selection bar appears as a highlighted (or differently colored) item in your work area. When you enter a window, the selection bar is on the top-most item in the work area. To move the selection bar simply put the cursor on the item you want and click once. THE ITEM IN THE SELECTION BAR IS THE ONE ON WHICH THE PROGRAM WILL OPERATE.

Scroll Bars

The scroll bars - one horizontal and one vertical - are located along the bottom and right-hand side of the window respectively. Located at each end of the scroll bars are scroll arrows. CLICKING AND HOLDING ON THESE ARROWS WILL CAUSE THE IMAGE IN THE WINDOW TO SCROLL. PICK THE ARROW POINTING IN THE DIRECTION YOU WISH TO MOVE. THE IMAGE WILL MOVE IN THE OPPOSITE DIRECTION. The white boxes in the scroll bars indicate the window's location. Your text can extend far outside the window's physical boundaries. Scroll enables you to move around within the boundaries of the text.

Work Area

The work area is the clear area bounded by the components described above. This is the area in which you will create text.

Cursors

Cursors are symbols that move in response to a pointing device (i.e., mouse). Five icons indicate the cursor location:

- The most common cursor is the **single-ended arrow**. To use it, move the point of the arrow to the desired location and press the mouse button (sometimes a double click is necessary).
- A **double-headed arrow** indicates that the cursor is on the thick border. Clicking and dragging the mouse causes a corresponding change in window size.
- When the cursor moves into an area where text can be entered, it converts into a **vertical I-beam**.
- A **vertical, flashing bar** will appear at the point text will start.
- An **hourglass icon** indicates that you must wait while the program completes an internal operation. Do not enter anything while this icon is onscreen.

Message Boxes

A message boxes is a pop-up window that appears in response to certain user selections. These boxes usually contain either

advice or caution. The message box may also offer options about proceeding with the selection you have made. Possible options include: OK, Cancel, Accept. To proceed, simply place the cursor on an option and click.

Dialog Boxes

A dialog box is another type of pop-up window that prompts for information in response to certain user selections. A dialog box is a means by which the computer accepts information, one field at a time. For example, one type of dialog box asks you to fill in three fields: title, date due, and link. Dialog boxes are provided so that you can add, change, or delete information in one field without affecting others.

Window Configurations

A window has three possible on-screen configurations.

- **Maximized** - Displays full-screen window.
- **Minimized** - Displays window with the size and shape you define using the thick border control method described above.
- **Iconic** - Displays window as an icon at the bottom of the screen.

Conclusion

This completes the section on using MicroSoft Windows. If you have a basic understanding of how Windows works, you are ready to progress to the next section: How KWS Works.

[The KWS Prototype provides three Microsoft Windows Utilities. These utilities are strictly optional; they have no interaction with KWS. To learn about these utilities, see Appendix B - Utilities.]

III. HOW KWS WORKS

Background

This section explains how to navigate within the KWS Prototype and access each type of KWS window.

Window Levels

The Knowledge Worker System consists of a hierarchy of windows. At the top of the structure is the TODO window. Beneath TODO is the Step window; beneath Step are the Attachments and DoIt windows, which are at parallel levels. Associated with the TODO and Step windows are Tips windows. The Notes window is associated only with the TODO window.

Following is information about each level of windows in KWS:

Help

Each window has a Help option in the Menu Bar. This option contains information on pertinent items for that window. This information has also been reproduced in Appendix A, at the rear of this guide.

TODO Window

When you first enter KWS, you will see with your TODO window. This window contains a chronological list of subtasks assigned to you by the scheduler (termed "scheduler subtasks"). Each of these subtasks has a Steps window associated with it.

Selection Bar

The TODO window contains a selection bar which, at system initiation, will be positioned on the first subtask that is due to be started but that has not been completed. To move the selection bar simply place the cursor on the desired subtask and click the left mouse button once. KWS WILL ALWAYS OPERATE ON THE HIGHLIGHTED SUBTASK.

Steps Window

To access the Steps window of a highlighted item, place the cursor on the selection bar and double-click the left mouse button. The Steps window associated with the subtask specified will appear. The name of the subtask will be in the window title box. The window contains a step-by-step list of instructions that describes what needs to be done to complete the subtask.

Return to TODO

There are two ways to return to the TODO window:

1) A double-click on the left mouse button will close the Steps window and display the TODO window.
2) A single click on the right mouse button will cause the TODO window to overlay all other open windows. This is the only operation for which the right mouse button can be used.

Associated Windows

Now that you know how to travel between the two main window levels, look at the associated windows. KWS has two types of associated windows: Notes and Tips. For instructions on how to add/modify/delete text in these types of windows, see Section III - How to Add/Modify/Delete Information.

Tips window

Tips is designed to encourage you to enter information relevant to a particular window. Every main-level KWS window has an associated Tips window. To see Tips, click once on the Tips option in the Menu Bar. Tips can be accessed only from the window with which it is associated.

Notes window

The Notes window meant to be a general notepad and is set up so that you can quickly enter notes to yourself. Notes is a system-wide window, that is, it is not tied to any subtask or step. There is only one Notes window and can be accessed only from the TODO window. Click once on Notes in the TODO Menu Bar and the Notes window will display.

DoIt & Attachments

KWS provides two important options at the Step level: DoIt and Attachments. These options allow you to attach programs and information associated with a single step. When you highlight a step, DoIt and Attachments will appear (on the Menu Bar) only if you have associated a program or text file with that step. Otherwise, DoIt and Attachments will appear in light gray text on the Menu Bar. For instructions on how to add/modify/delete text, see Section III - How to Add/Modify/Delete Information.

DoIt

DoIt allows you to simulate program execution. To access an existing DoIt, click once on DoIt in the Menu Bar. A drop-down menu of programs associated with the highlighted step will appear. Click once on the name of the program you wish to execute. A message box will appear and prompt you to run the program or cancel the operation. Click once on the option you desire. If you choose to proceed, program execution is simulated and its output is placed in a file that then appears in the DoIt menu. If you cancel the operation, you are returned to the Steps window.

NOTE: The DoIt option is included in the Prototype for

demonstration purposes only; the Prototype implementation of DoIt does not have true execution capability.

Attachments

Attachments is accessed in a manner similar to the DoIt option. Attachments, however, displays a text file that contains information relevant to a particular step (i.e., another step, a subtask, correspondence, requisitions, etc.)

NOTE: In the Prototype, only text (ASCII) files can be displayed through the Attachments option.

Conclusion

This section completes the general description of the Knowledge Worker System Prototype organizational structure. If you feel comfortable with your understanding of the system structure, continue to the next section: How To Add/Modify/Delete Information.

IV. HOW TO ADD/MODIFY/DELETE INFORMATION

Background

This section describes how to alter existing KWS data. KWS uses dialog boxes to add, modify, and delete information. Dialog boxes are a structured means of prompting the user for information. In the text below, the four types found in KWS are described: Edit TODO dialog box, Edit Step dialog box, DoIt dialog box, and an Attachments dialog box.

TODO Window

The TODO window contains both scheduler subtasks and personal subtasks. **"Scheduler subtasks"** are derived from tasks that have been pre-assigned to you. KWS will not let you alter scheduler subtasks in any way unless you have scheduling permissions [see the Scheduling Supplement to this Guide for more information]. The only exception is that you can mark a scheduler subtask finished/ unfinished using FixIt! (see Appendix A - HELP).

"Personal subtasks" are those that you add to the system. The procedures to add, modify, and delete your own personal subtasks are described below.

Adding a Personal Subtask

To add a subtask, begin at the TODO window. Click on FixIt! and a drop-down menu will appear. Move the cursor to Add and click once. The Edit TODO dialog box will appear.

Edit TODO Dialog Box

The Edit TODO dialog box contains four data fields (Title, Date Due, Amt Time, and Link) and an Accept/Cancel option. Use the Tab key to move from one data field to the next. Depress the Shift and Tab keys together to move backwards through the fields. Or use the mouse button to select a specific field.

The data fields are:

Title - Accepts any combination of characters that fits within the length of the box.

Date Due - Enter in the form of day,month,year (e.g., 23OCT89)

Amt Time - Indicates how long you have worked on the associated subtask. You may wish to use this information later to update the duration data for the subtask.

Link: Enter the name of the subtask with which the current subtask is associated.

The Accept/Cancel option prompts you to either: Accept the data displayed on the screen, or to Cancel the dialog box without modifying the ToDo list. The data will only be entered into KWS if you choose Accept. Once you click on Accept or Cancel, the dialog box closes and you return to the TODO window.

Modifying a Personal Subtask

This procedure is similar to adding a subtask. From the TODO window, pick the subtask you want to modify. [Be sure the subtask you want to change is highlighted.] Click on FixIt! and then on Modify (in the drop-down menu).

When the Edit TODO dialog box appears, change any data fields you wish to modify. Then click on Accept to implement the changes, or on Cancel to abort the operation without modifying the ToDo List. Either choice will return you to the TODO window.

Deleting a Personal Subtask

From the TODO window, highlight the subtask you want to delete. Click on FixIt! and then on Delete (in the drop-down menu). A message box will appear and confirm your deletion request. The box contains a Yes/No option. Click on Yes and the subtask will be deleted, or No to cancel the operation. [Note: Subtasks can be restored by choosing to Add a Subtask & reentering the fields called for via the dialog box. Steps and associated items do not have to be reentered.]

Steps Window

After creating or receiving a subtask, you are ready to add the steps or the "how-to information" about the subtask. Break the subtask down into the sequence of steps required to perform it.

Adding a Step

To add a step, begin by highlighting the appropriate subtask in the TODO window. Double-click on the subtask. The Step window will appear on the screen. Click on FixIt! and then on Add. The Edit Step dialog box will appear.

Edit Step Dialog Box

This dialog box contains one data field (Step Title), two list boxes (Attachments and DoIts), and an Accept/Cancel option.

Step Title: This text field accepts any combination of characters.

Attachments: This list box for ASCII (text) files accepts any type of text.

DoIt: This list box for executable programs simulates

the capability to store and run a program.

The Accept/Cancel option is described above under "Edit TODO Box."

To add a step without Attachments or DoIts: Simply enter the step [in the Step Title field] and click on Accept.

NOTE: **Do Not** number the steps! KWS will number the steps after you create each one. Once you click on Accept, the numbered step will be inserted into the list of steps.

Adding Additional Steps:

When you add steps (that is, when one or more steps are already present), the new step will be inserted after the one that is currently highlighted. Therefore, the first rule in adding additional steps is to always highlight the step that will precede the step you wish to add.

For example:
To add a step between #3 & #4, highlight #3 then click on FixIt! and then on Add. The new step will become #4 & the subsequent steps renumbered.

When you add several steps in a row,you will note that KWS automatically moves the highlight down to the most recently-added step.

Adding An Attachment

To add an Attachment, begin at the Edit Step box. Click on Add Attachment. Another dialog box (Edit Attachment) will appear.

Edit Attachment Dialog Box

The Edit Attachment box has two data fields (Title and File name), a directory display, a list box, Subtask option, and an Accept/Cancel/Delete option. The following items are included in this dialog box:

Title - Enter the name by which to refer to this attachment (e.g., Checklist 221-1).

File name - Enter the name of the text file in which the attachment is stored.

Directory display - Enter the path for the attachment.

List box - This feature allows you to search for existing files. The initial display includes [..] and the drives available.
Double-clicking on [..] will move you up one level in

your system directory hierarchy.
Double-clicking on a drive tells KWS to search that drive.

Subtask - If you are attaching a subtask click on this option.

To Accept, Cancel, or Delete, click on the appropriate box. You will be returned to the Edit Step box.

Adding a DoIt

To add a DoIt, begin at the Edit Step box. Click on Add Attachment. Another dialog box (Edit DoIt dialog box) will appear.

Edit DoIt Dialog Box

This box has the same elements and operates the same way as the Edit Attachment box with one exception. In place of the Subtask field, the Edit DoIt box contains an field titled "Executable"

Executable - Check this box (by clicking on it) to tell KWS to simulate running the program.

[If this box is not checked, when DoIt is selected from the Step window, the text of the specified file will list.]

Modifying a Step

Highlight the desired step, click on FixIt! and then click on Modify, and the Edit Step box will reappear. The final sequence in each of the following actions is to click on Accept or Cancel.

Modifying the Step Title: Be sure the cursor is on the Step Title box and alter the text as desired.

Modifying Attachments and DoIts: Double-click on either the attachment or DoIt that you wish to change and you will be placed in the appropriate dialog box. Alter the text as desired

Moving Attachments and DoIts: Between the Attachments and DoIt list boxes (in the Edit Step box) are two double-headed arrows. You can change a DoIt to an Attachment (or vice-versa) in the following way:

- o Highlight a DoIt (in its list box)
- o Click on the appropriate arrow to move the item selected from the DoIt box to Attachments.

Deleting a Step

Highlight the desired step, click on FixIt! and then Delete. The Delete message box will appear to confirm that you really mean to delete the step in question. Click on Yes only after ensuring that you have selected the step you intend to remove.

Tips & Notes Windows

To enter information into these windows, KWS provides text windows. Simply click on Tips or Notes in the Menu Bar. The specified window will appear.

Tips

Once the Tips window opens, follow this sequence to add or modify text:

1) Click on FixIt!, a drop-down menu will appear.
2) Click on Editing enabled.
3) Enter/alter the text, as desired.
4) Before exiting the window, SAVE YOUR WORK!
   a) Click on FixIt!
   b) Click on Save.
   Your work is now saved.
5) Exit the window by the standard methods.

Notes

Once the Notes window opens, follow this sequence to add or modify text:

1) Enter/alter the text, as desired. [Editing is already enabled in this window.]
2) Before exiting the window, SAVE YOUR WORK!
   a) Click on FixIt!
   b) Click on Save.
   Your work is now saved.
3) Exit the window by the standard methods.

Conclusion

This section documents the basics of KWS: How to add, modify, and delete information. You are encouraged to use these basic operations and enter data relevant to your assigned responsibilities. Experiment with KWS so that you can provide feedback about the usefulness (or not!) of the Prototype's functionality. Your comments/ suggestions/ criticisms are solicited!

APPENDIX A - HELP

EVENT.HLP

The SYSTEM SCHEDULER window is a list of events, presented in chronological order.

Events that are late are marked in red; events not associated with a date (e.g., on-going events) are marked in blue.

To view the tasks associated with a event, double click on that event.

MENU ITEMS

Help - Opens this window.

FixIt! - Presents a submenu that allows you to: Print the window, mark an item Finished, Add/Delete/Modify event date, invoke the Scheduler, or update the KW's ToDo lists. [The FixIt! menu is described in more detail below.]

Tips - Opens a window with general information associated with the event list.

Notes - Opens a window that can be used as a personal notepad.

Windows - Presents a submenu of all KWS windows you have currently have open. You can jump directly to any window on this list by clicking on that item.

More about the FixIt! submenu

Print

Print the contents of this window on the default printer.
[The default printer is defined using the Windows Control Panel thru MS-DOS Executive. See Microsoft Windows User's Guide, Chapter 7 for more details.]

Finished

Marks an item "finished." Places >> in front

of the event. (Acts as a toggle: Changes between "unfinished" & "finished" by removing or adding the >> mark, as appropriate.)

Add

Opens a dialog box that prompts you for the details needed in order to add a new event to the KWS system.

Delete

Deletes the currently-selected event from the KWS system (the one printed in white on a black background).

Modify

Opens a dialog box that prompts you to change information associated with an existing event.

Schedule

Opens a dialog box which allows you to change the due date associated with an event. If you change the due date by chosing Accept, the KWS scheduler will be invoked and apply the new due date to all tasks involved in the event. This differs from the Modify option. Modify will change a date without changing the dates of associated tasks.

Update KW's ToDo

Invokes the KWS scheduler and generates a new ToDo list for each Knowledge Worker. For each Knowledge Worker, the ToDo list generated here will be combined with their personal ToDo list.

STEP.HLP

The STEP window is a list of steps, presented in the order in which each should be performed.

Each step may have programs or supporting documentation associated with it. When you choose a step (by clicking on it), KWS will indicate if there are programs or supporting documentation by darkening the "DoIt" or "Attachment" choices respectively.

To execute a program select "DoIt" in the menu bar. A drop down menu will show you the programs that can be executed. Click on the one you want. KWS will confirm that you want to run the program before execution begins.

To view an attachment, select "Attachment" in the menu bar. You will see a list of attachments. Click on the one you want to view, and a new window will open showing you that document.

MENU ITEMS

Help - Opens this window.

FixIt! - Presents a submenu that allows you to: Print the window or Add/Delete/Modify steps [The FixIt! menu is described in more detail below.]

Tips - Opens a window with general information associated with the Step window.

DoIt - Presents a drop down menu of programs associated with the current step. If there are no programs this choice will be displayed in light gray.

Attachments - Presents a drop down menu of supporting documentation associated with the current step. If there is no supporting documentation, this choice will be displayed in light gray.

More about the FixIt! submenu

Print

Print the contents of this window on the

default printer.
[The default printer is defined using the Windows Control Panel thru MS-DOS Executive. See Microsoft Windows User's Guide, Chapter 7 for more details.]

Add

Opens a dialog box that prompts you for the details needed in order to add a step.

Delete

Deletes the currently-selected step (the one printed in white on a black background).

Modify

Opens a dialog box that prompts you to change information associated with an existing step.

TASK.HLP

The TASKS window is a list of tasks associated with a particular event, presented in chronological order.

Tasks that are late are marked in red; tasks not associated with a date (e.g., on-going tasks) are marked in blue.

To view the subtasks associated with a task, double click on that task.

MENU ITEMS

Help - Opens this window.

FixIt! - Presents a submenu that allows you to: Print the window, mark an item Finished, Add/Delete/Modify a task date, or invoke the Scheduler [The FixIt! menu is described in more detail below.]

Tips - Opens a window with general information associated with the task list.

Notes - Opens a window that can be used as a personal notepad.

Windows - Presents a submenu of all KWS windows you have currently have open. You can jump directly to any window on this list by clicking on that item.

More about the FixIt! submenu

Print

Print the contents of this window on the default printer.
[The default printer is defined using the Windows Control Panel thru MS-DOS Executive. See Microsoft Windows User's Guide, Chapter 7 for more details.]

Finished

Marks a task finished. Places >> in front of the event. (Acts as a toggle: Changes between "unfinished" & "finished" by removing or adding the >> mark, as appropriate.)

Add

Opens a dialog box that prompts you for the details needed in order to add a new task

to the KWS system.

Delete
: Deletes the currently-selected task from the KWS system (the one printed in white on a black background).

Modify
: Opens a dialog box that prompts you to change information associated with an existing task.

Schedule
: Invokes the KWS system scheduler. The system scheduler will display all subtasks associated with an event and allow you to make scheduling changes.

SWINDOW.HLP

This window contains a list of all subtasks associated with a particular event. The subtasks are presented in chronological order.

In this window you may perform what-if analysis with the schedule. No changes will be made to the schedule unless you answer "YES" to the dialog box which appears when you try to close the window.

Subtasks that are late are marked in red. Items not associated with a date (e.g., on-going subtasks) are marked in blue.

There are five columns in this window. The first column shows the due date of a subtask (resulting from changes made during the what-if analysis.) The second column shows the due date as it currently is in the master schedule.

The third column shows the number of days needed to complete the subtask (as specified when the subtask was entered.) The fourth column shows how many days are actually left before the subtask is due.

The fifth column is the subtask title.

MENU ITEMS

Help - Opens this window.

FixIt! - Presents a submenu that allows you to: Print the window or Modify a subtask. [The FixIt! menu is described in more detail below.]

Notes - Opens a window that can be used as a personal notepad for items on your ToDo list.

Windows - Presents a submenu of all KWS windows you have currently have open. You can jump directly to any window on this list by clicking on that item.

More about the FixIt! submenu

Print

Print the contents of your this window on the default printer.
[The default printer is defined using the Windows

Control Panel thru MS-DOS Executive. See Microsoft Windows User's Guide, Chapter 7 for more details.]

Modify

Opens a dialog box that prompts you to change information associated with the currently selected subtask. If you signify a change (by clicking on "Accept" in the dialog box), the scheduler will calculate the impact on the other subtasks in the event.

TODO.HLP

The TODO window is a list of subtasks, presented in chronological order.

Subtasks that are late are marked in red; items not associated with a date (e.g., on-going subtasks) are marked in blue.

To see subtasks that have been marked "finished" scroll to the very top of the window. Finished subtasks are preceded with this mark: ;

Any items that you add ("personal subtasks") are preceded with this mark: *

To view the steps associated with a subtask, double click on that subtask.

For quick access to this window from other KWS windows, press the right-hand mouse button.

MENU ITEMS

Help - Opens this window.

FixIt! - Presents a submenu that allows you to: Print the window, mark an item Finished, Add/Delete/Modify personal subtasks, or Change your user profile. [The FixIt! menu is described in more detail below.]

Tips - Opens a window with general information associated with the ToDo list.

Notes - Opens a window that can be used as a personal notepad for items on your ToDo list.

Windows - Presents a submenu of all KWS windows you have currently have open. You can jump directly to any window on this list by clicking on that item.

More about the FixIt! submenu

Print

Print the contents of your ToDo list on the default printer.
[The default printer is defined using the Windows Control Panel thru MS-DOS Executive. See Microsoft Windows User's Guide, Chapter 7 for more details.]

Finished

Marks an item finished. Places >> in front of the subtask. (Acts as a toggle: Changes between "unfinished" & "finished" by removing or adding the >> mark, as appropriate.)

Add

Opens a dialog box that prompts you for the details needed in order to add a personal item to your ToDo list.

Delete

Deletes the currently-selected ToDo item (the one printed in white on a black background). Only personal subtasks can be deleted thru this option. (If the item currently selected is not a personal subtask, Delete will be light gray, signifying that the Delete choice is not available.)

Modify

Opens a dialog box that prompts you to change information associated with an existing ToDo item. Only personal subtasks can be changed thru this option.
(If the item currently selected is not a personal subtask, Delete will be light gray, signifying that the Delete choice is not available.)

Change Profile

Presents the original login screen & allows you to change your user profile. You can change your name, storage location of your KWS data, & user id. (If you have the scheduling password, use this option to enable/disable scheduling privileges.)

Schedule

Opens the master schedule window.
(If you are not allowed to change the master schedule, this option will be gray, indicating that it is not available.)

NOTE.HLP

This window contains a text document. You may change the contents of the document by clicking on the "Editing enabled" choice under FixIt!.

MENU ITEMS

Help - Opens this window.

FixIt! - Presents a submenu that allows you to: Print the window, Enable editing, Cut/Paste/Copy/Delete text, or Select all the text in the window. [The FixIt! menu is described in more detail below.]

More about the FixIt! submenu

Save

Saves any changes made to the document shown in the window. If no changes have been made, this choice will be displayed in light gray.

Print

Print the contents of this window on the default printer.
[The default printer is defined using the Windows Control Panel thru MS-DOS Executive. See Microsoft Windows User's Guide, Chapter 7 for more details.]

Editing enabled

Controls whether or not the contents of the window can be modified. When editing is enabled a check mark appears in front of this choice.

Cut

Removes the selected text from the document and places it on the Window's Clipboard.

Paste

Inserts the contents of the Window's Clipboard into the document, at the cursor.

Copy

Places the selected text from the document into the Window's Clipboard without removing it from the document.

Clear

Removes the selected text from the document without placing it on the Window's Clipboard.

Select All

Selects all the text in the document for one of the previously discussed operations.

APPENDIX B - UTILITIES

Background

Three Microsoft Windows utilities have been included in the KWS Prototype: Calendar, Clock, and Cardfile. The icons that represent these utilities appear in the lower-left corner of the monitor screen.

A brief explanation of what each utility does is included here. For detailed information (i.e., directions on their use), see the Microsoft Windows User's Guide, (Version 2.0), Book 2: Page 39 (Calendar),
Page 69 (Clock),
Page 17 (Cardfile).

Clock -

Clock is the simplest utility so we will start with it. This standard clockface displays the current time, as set by your system. Clock can be shaped to any size, placed any where on the screen, and always displays the correct time, even it appears as an icon.

Cardfile -

Cardfile is a filing program that you can use to keep track of names, addresses, phone numbers, directions, or anything else for which you want quick access. Cardfile is formatted as a set of index cards, sorted in alphabetical order.

Calendar -

Calendar is intended as a desktop calendar, appointment book, and alarm clock. You can maintain more than one Calendar file and keep track of appointments for more than one person. Calendar provides the ability to set an alarm (e.g., to remind you of an appointment). Calendar offers two views of time. In the **Day** view, you can enter, display, or edit your appointments for a particular day. In the **Month** view, you begin by selecting the day for which you want to see appointments.

APPENDIX C - GLOSSARY

Knowledge Worker: A professional whose major responsibility is the collection, dissemination, analysis, and presentation of information.

The following KWS terms are arranged in descending hierarchical order.

Event: An activity which is associated with a product, e.g., POM Lock; or the production of the Congressional Budget Book. Events are composed of tasks. An event has a date that can only be changed at the highest levels of the organizational hierarchy.

Task: An activity with a definite ending date and for which a supervisor is responsible. Tasks are composed of subtasks.

Subtask: An activity which is a sub-component of a task. A knowledge worker's ToDo List is composed of subtasks. Subtasks have due dates, precedence, and duration.

Trigger: A precedent subtask that is completely passive but which occur in order to initiate a task or another subtask, e.g., "Receive AG Vol II" must be complete before "Analyze AG Vol II" can begin.

Step: How-To Information; one of a series of actions necessary to perform a subtask. Steps are ordered sequentially. Additional information may be associated with steps, e.g., another subtask or step; correspondence; an executable program; the output of an report, etc.

Attachments: Any number of items that may provide additional information about how to perform a step. Examples of attachments include primarily supporting documentation, e.g., memorandums, letters, routing slips, or reports from a previous year. Attachments can also be another subtask or step.

SCHEDULE FOR TASK 2 & 3 SUBTASKS

SUBTASK 2.1
31 JUL CERL provides tasks for schedule to GT
11 AUG CERL & GT review tasks

SUBTASK 2.2
13 SEP Scheduler developed
CERL comments to GT

SUBTASK 2.3
09 AUG User Interface developed
21 AUG CERL comments to GT

SUBTASK 2.4
23 AUG File transfer capability b/tw PAX & KW machine
28 AUG CERL comments to GT

SUBTASK 3.1
20 SEP Demo ready for CERL review
25 SEP CERL comments to GT

SUBTASK 3.2
13 OCT User's Guide at CERL for review
16 OCT CERL comments to GT
23 OCT User's Guide incorporates CERL's comments

SUBTASK 3.3
25 OCT Install Prototype in CEEC-P
26 OCT Training for Govt personnel

SUBTASK 3.4
26 OCT Begin tech support period
22 DEC Technical support ends

SUBTASK 3.4
13 NOV Participate in 3-day workshop at Fusion Center

SUBTASK 3.2
10 NOV Updated version of Design Document (if necessary)
10 NOV Tech Guide ready for CERL review
27 NOV CERL comments to GT
15 DEC Tech Guide incorporates CERL's comments

SUBTASK 3.2
29 NOV Outline of Lessons Learned topics to CERL
08 DEC Lessons Learned ready for CERL review
22 DEC CERL comments to GT
12 JAN Lessons Learned incorporates CERL's comments

18 JAN Contract ends

KNOWLEDGE MANAGER PROTOTYPE DESIGN

Prepared Under Contract
DACA88-88-D-0020/0008

Task 1-5
Detailed Design Of Knowledge Manager Prototype

Submitted to:

Department of the Army
CONSTRUCTION ENGINEERING RESEARCH LABORATORY
Champaign, Illinois

Prepared August 14, 1989 by:
GEORGIA INSTITUTE OF TECHNOLOGY
Atlanta, Georgia

Table Of Contents

Goals

The purpose of the Knowledge Manager (KM) system is to preserve institutional knowledge and assist action officers in the performance of their daily tasks. KM must be able to provide answers about what tasks to perform, as well as how and when to execute them. The system must be able to maintain and display a dynamic schedule of tasks to be performed. The KM will free the knowledge worker (KW) from repetitive tasks by automating as many steps as possible.

The goal of this prototype to provide a useful subset of these KM functions, as well as to provide a platform on which to study the best methods for implementing the production version of the Knowledge Manager.

Overview

Two sections make up this report. The first section provides a general overview of the system. In this section the fine points of the system are only briefly described. In the second section, the interaction of the components of the Knowledge Manager is described in greater detail.

The design of the prototype Knowledge Manager is based on the object-oriented language, Actor, which runs under Microsoft Windows. In an object-oriented design, data and the actions that can be performed on that data are combined in an entity called an object. All objects that share common attributes are grouped together into "classes". A object which is a specific member of a class is referred to as an "instance".

Associated with any class there are a set of actions which any object in that class knows how to perform. These actions are called "methods". Writing methods for a class is how one programs in an object-oriented language.

Classes are organized hierarchically, from the most general to the most specific, with each class inheriting all the capabilities and behaviors of the classes from which it is descended. Thus, when new classes are defined, objects in the new class behave exactly like objects in the parent class; the programmer need only add methods to the new class to specify any new or different behaviors for objects in the new class.

In keeping with the object-oriented design of the Knowledge Manager, the interaction between KM components will be described in terms of the messages to which each component can respond. This general view of the system allows most of the effort in the prototype to be focused on the design of the system as it would perform on an ideal platform.

This approach will minimize the effort in implementing the production version of the Knowledge Manager. Since the system is described in terms of its reaction to messages, only the lowest level of the message-processing functions (those that deal with the operating system) will need to be modified for a different operating system.

From a general standpoint, the interaction of an object, the windows in which it is displayed, and the file that holds its data is transparent to the user. This window-object-data synthesis provides the user with the illusion that the data can be changed directly and in real-time.

For example, when a user changes the percent-complete value in one window, it appears to the user that the value of interest was selected and a new value inserted. What really happens is this: The user changes the data on the screen, the window sends a message telling the object that it has been changed, and the object performs some operating system dependent operation (discussed in detail in the Objects section) and the data is stored in a file.

The User Interface

In order to provide the user with consistent information about the status of the system, several basic themes will be followed throughout the user interface.

The use of colors

Standard colors will be used to provide the user with instant information about the status of information on the screen. So that a KW sees at a glance the status of all the subtasks on the ToDo list, the colors red, green, and yellow will be employed consistently. In particular, red will be used to indicate a situation that requires immediate attention, for instance, subtasks which are late. Yellow will be a warning color and used to draw the user's attention to some important - but not yet critical - situation. Green will indicate that an item is within normal operating constraints. The normal background color for X will be blue; normal text will be white; and places where the user may double click the mouse and link to other X windows will be black.

These colors will be modifiable, however, this capability will not be given to the user; rather, it will allow experimentation with colors during prototype evaluation. The user will not be allowed to customize the colors. Color standardization is necessary to prevent a KW from accidentally misreading a situation if evaluating from a workstation where another user implemented a different color scheme.

Window types

There are five basic types of windows the user will see. These are the: 1) ToDo list, 2) Schedule window, 3) Task window, 4) Subtask window, and 5) Attachment window. Each window has slightly different properties that enable it to present its particular kind of information.

ToDo list window

The ToDo list window displays the knowledge worker's daily list of tasks (actually subtasks). This list shows each task that can be started and is sorted in order of due date. A ToDo window can process hypertext links, has knowledge of dates, and allows the KW to edit private (user-discretionary) tasks. The ToDo window is the point where a KW will initially access the KM system. It is attached to all other windows in the KW's knowledge base. The ToDo window may always be brought to the front of the on-screen windows by pressing the right-hand mouse button. The menu bar of the ToDo window contains a "News" button. When the KW selects this button, KM traverses a link to an attachment containing the daily system news.

The ToDo window contains one line for each task the KW needs to perform. A more detailed description of a particular item may be found by double clicking the mouse while pointing to this line; this action will bring up a subtask window.

A user may enter personal subtasks in the ToDo window. In the prototype the personal subtasks that recur periodically must be entered repeatedly. Personal subtasks will show up in the ToDo list just like a subtask assigned to a KW by the scheduler, except they will have a special symbol in the first column to aid the user in distinguishing them.

Once a subtask is marked complete, it is no longer displayed in the ToDo list. However, if a KW scrolls the ToDo window backwards from the current day, a historical list of all subtasks ever scheduled can be reviewed along with completion dates, etc. Likewise, the KW can scroll the ToDo list forward in time to view items that will appear on the list in the future.

Colors will be used in the ToDo list to draw the KW's attention to special conditions. If a due date has been changed since the last time a KW looked at the ToDo list, the item will be highlighted in yellow. Also, any subtask which is currently late will be highlighted in red.

Schedule window

The schedule window is very similar to the ToDo list window. This window will generally be the entry point to the Knowledge Manager system for users at the supervisory level and above.

This window allows the user to manipulate the schedule and gain access to the task-level information. This window is described in greater detail in the schedule section.

Task window

The task window will look identical to the subtask window (see below). The only difference between the two is their level of decomposition. A task window decomposes into subtasks; a subtask window decomposes into steps.

Subtask window

The subtask window displays the steps that make up the subtask, and associated information. This window can process hypertext links, is editable (that is, the KW can modify the subtask description), can create/delete attachments, and can link to or execute attachments. From this window the user can link to the task-level description of the subtask and display this description in a task window. When the KW first opens a subtask window, the Knowledge Manager will position the selection bar at the first uncompleted step.

The menu bar of a subtask window contains three choices of particular interest: "Info" and "DoIt" and "Attachments". The "Info" menu button causes a window to pop up with information about the subtask, such as, percent completion, a short description, etc. The "DoIt" menu choice presents the KW with a submenu containing any executable items as well as their related files. By selecting an item on the "DoIt" submenu, the user directs Knowledge Manager to automatically perform the particular step. This automatic execution capability will be simulated in the prototype. The "Attachments" menu button will bring up a submenu which contains a list of items attached to the step currently selected.

Usually each step will have at least one item attached to it, i.e., a more detailed description of that step. If the user double clicks the mouse on a step, this link will automatically be traversed, and the associated description will be shown in either attachment window or another subtask window, depending on the type of link. Hypertext links in a subtask window may only be attached to steps. By using the "Attachments" submenu, a large number of links can be attached to each step.

The Knowledge Manager will keep track of the time spent in performing each subtask in order to assist the KW in estimating the amount of time required for completion. The Knowledge Manager will use a simple heuristic in order to estimate the time a KW spends per subtask. This heuristic is: Any time a subtask window or any of its child windows is active, the KW is assumed to be working on the subtask. The system will display the total number of minutes spent on a subtask and allow the KW to modify this number in order to correct inaccuracies.

Attachments

Attachment windows are simply ASCII file listers. These windows understand only one command: close. This window type represents the terminal type of the KM system; no hypertext links may originate from one of these window types.

Hypertext links

The type of data contained in an attached file will be associated with the link. For instance, a hypertext link may indicate that the attachment contains a document; another link may indicate that the attachment is an executable program. This information will provide the system with the ability to make intelligent decisions about what action to perform when a user activates a hypertext link. In the prototype, the only types of hypertext links the KM will be able to process are subtask links, attachment links, and execution links (which will only be simulated). When the user commands the KM to traverse a link using one of the methods described previously, the KM determines link type, creates an appropriate window type, and loads it with the data attached to the link. In the case of an executable link, the KM would run the designated program and pass it the attached data file.

Organization Of Data

This section describes the flow of the data through the Knowledge Manager system from the master files on the Programming, Administration, and eXecution system (PAX) down to the user files on each knowledge worker's PC. For purposes of clarity, some detail about the actual file formats is not included here. The complete description of the format for each file can be found in Appendix I.

The data will be stored on each KW's personal computer in its final form. That is, the scheduler will have been run on it and all hypertext links are established.

When a subtask is reassigned to a different KW, all the associated files will be copied to a floppy disk that can be transferred to the computer of the KW to which the subtask is reassigned.

ToDo list files

On each PC there will be one file containing the ToDo list. This file is the base of the Knowledge Manager system. Starting from this file, any other file can be reached by traversing the links. Basically, this file contains a list of each subtask assigned to the KW and its due date. The ToDo file is produced by combining the output of the scheduler with the KW's private schedule. Details of how to perform a subtask are found by traversing a link associated with a subtask to the associated description.

In addition, a knowledge worker can add any number of user-discretionary subtasks to the ToDo list. These subtasks are considered private to the individual knowledge worker and not moved back to the central archive. All files associated with private subtasks will be stored under a separate directory on the user's hard disk. This storage strategy will allow the user to find the files easily for backup purposes and will prevent naming conflicts with Knowledge Manager subtasks.

Schedule files

The master schedule file will reside on the PAX system. Additionally, a copy of the master schedule will reside on each KW's computer. These multiple copies are kept synchronized by the messaging system (see MESSAGES).

In the production version of the Knowledge Manager, the joining of the data from all the task objects should be done by a database server.

Task files

The task description file is interesting in that it is viewed differently, dependent upon the level in the system from which it is accessed. At the system level, the task description file is the fundamental entity for the scheduler. From the knowledge worker's point of view, this file is merely an attachment which gives some general information about a subtask. From a supervisor's point of view, it is the entry point of the system and ties assignments together.

Subtask files

There is one file for each subtask on the ToDo list. This file contains the general description of the subtask as well as its due date, a list of steps involved in performing this subtask, and links to any attachments associated with the subtask.

Each subtask file includes a link to its task description file. This file contains general information about the task of which this subtask is part. System wide, only one task file exists for each task; each system-wide file is linked to all of its subtask files.

Attachment files

An attachment is a file of any format that contains some information (text, documents, programs, data) which might be helpful to the knowledge worker. Schedule, Task, or Subtask files can have attachments. Part of an attachment link field contains information about how to operate on that information. In the prototype, only ASCII text information can be manipulated, however, any other attached information will be carried through the system.

Hypertext links

The hypertext links are implemented by using DOS as a database system. Each file contains one record of data, with the file name being the key to the record. A hypertext link can be established by adding the name of one file to another file. This very general database system allows the hypertext links to be maintained across the diverse operating environments present within the Knowledge Manager system. In the production version, this system could be easily implemented to run under a database server.

Scheduler

Overview

Analysis of the nature of Planning, Programming, Budgeting, and Execution System (PPBES) tasks determined that a classical critical path method (CPM) scheduler is not necessary in the Knowledge Manager system. Specifically, knowledge workers' tasks are related to each other through externally-fixed ending dates. In typical CPM schedulers, ending dates are calculated by arranging tasks to complete a process as quickly as possible. In contrast, in PPBES it is possible for all tasks to have slack time. This structure seems to preclude CPM.

The prototype concentrates on the scheduling subsystem functionality rather than specific scheduling algorithms. In its daily operation the Knowledge Manager scheduler will provide access to a list of subtasks that should be in progress. The prototype will keep track of a knowledge worker's schedule and block out times when a KW will be unavailable to work on a subtask. Additionally, the system will provide the capability to transfer responsibility for a subtask from one KW to another. In the prototype, when subtasks are reassigned the associated files will be copied to a floppy disk for manual transfer.

The scheduler window is the first window viewed by a supervisor. This window will display all the tasks in the knowledgebase in chronological order. From this window a user can add or delete tasks, bring up a task description window, change task due dates, and do what-if analysis on the schedule.

Changing the schedule

The ability to do "what-if" analysis on the schedule will be restricted; only users with the proper authorization can experiment with changing task due dates. Furthermore, only authorized users can apply these changes to the master schedule. A user may restore the schedule from the master schedule at any time during a what-if analysis session.

When the user changes a due date for a task, the scheduler will run through the predecessor list and adjust related dates. If the scheduler finds any task that does not have enough working days for completion, it will be flagged as late.

When a user with sufficient authority chooses to apply the date changes to the master schedule, the scheduler broadcasts a message to each KM user's machine. Users who are currently on-line will receive this message that indicates what changes have been made to the schedule. Other users will receive a schedule-update message upon subsequent login to KM.

Adding or deleting tasks

Only authorized users can add or delete a task. If a user does not have this privilege, menu buttons will be grayed to indicate that this option is not available.

When a task is added or deleted, a taskAdded or taskDeleted message is sent to users currently logged into KM. Other users will receive a similar message upon subsequent login to the system.

Accessing task window

When a user double clicks on a task line in the scheduler window, the associated task window is opened on the screen.

Messages

The TASK SCHEDULE master database file contains information which is also duplicated in various files residing on the KW's personal computers. Whenever this information is changed, either by a scheduling change or through the normal day-to-day operation of the system by the KWs, a message must be sent to all computers containing copies of the changed information in order to keep the data consistent throughout the Knowledge Manager system. This type of message is called a "CHANGE MESSAGE".

A second type of message called a "LOCK MESSAGE" is used to insure only one person at a time attempts to make schedule changes.

PC format of messages

A CHANGE MESSAGE consists of a file containing the information necessary to edit a field in the TASK SCHEDULE database, which resides on PAX. Specifically, this information is:

KWID | Task # | Subtask # | Field name | New Value

Where KWID is the identification number of the Knowledge Worker sending the message, Task # and Subtask # are used as keys to look up the particular record being changed, and Field name is the particular field that the New Value will replace. The CHANGE MESSAGE file contains one line for each message; this ensures that messages are processed in the order they are generated.

A LOCK MESSAGE consists of a file containing the KWID of the Knowledge Worker requesting the schedule file lock.

PAX format of messages

The CHANGE MESSAGE file on PAX is a data base containing one record for each message that has not been delivered. Two additional fields in each message record are the KWID of the intended recipient and a time stamp that ensures messages are processed in the order they are sent.

If the master schedule is locked, there will be a file called "SCHEDULE LOCK" containing the KWID of the owner of the lock and a time stamp telling when the lock was applied. The schedule may only be written to PAX by the person who owns the lock.

Flow of messages to and from a KW

A change message is generated whenever a KW edits a subtask information field that is also a field in the TASK SCHEDULE. At the time of the change, a record is written to the CHANGE MESSAGE file.

The next time the KW connects to PAX, the CHANGE MESSAGE file is uploaded and the changes are applied in the order they were generated. Next, the PAX CHANGE MESSAGE database file is searched for any messages destined for the KW. If present, these messages are downloaded to the user's PC and applied to the local copy of the files.

Flow of messages to and from a supervisor

A supervisor must place a lock on the master schedule before making any changes to the schedule. To do so, the supervisor's computer connects with PAX checks to see if a lock is already in place. If no lock is in place, one is created using the supervisor's KWID. Whether or not the schedule is locked, a copy is downloaded to the supervisor.

After the changes are made to the schedule, a series of change messages are sent to PAX to change the master schedule. Change messages are sent rather than uploading the schedule; this allows KWs to continue updating their reporting information even while the schedule is locked.

When the change messages are applied to the master schedule, a second message will be generated for each message sent by the supervisor. This message will be addressed to the particular KW assigned to any subtask that was altered and will contain the changes to be applied to the KW's local copy of the data.

If the change message alters the assigned_to field for a subtask, then a message will be sent to both KW's involved.

Database synchronization

In any system that keeps the same data in different places, extra care must be taken to insure that all copies of a piece of data are updated together. The methods described in this subsection ensures that this synchronization takes place.

The master schedule is never written to on a record basis, rather the data is entered one field at a time. For most fields there is only one person with the authority to make modifications, namely, the KW assigned to the subtask to which the field applies. Supervisors have overlapping authority to change fields, therefore, they must lock the master schedule before modifying it so that other supervisors can not make simultaneous changes. The fields that supervisors can change and those that KWs change are mutually exclusive; there is no possibility of conflict regarding changes between supervisors and KWs.

Detailed Object Descriptions

Task

Description

Task objects will be created from the schedule window. A task object accepts messages from the user and performs the specified actions on a task description file. If any message causes a change in the contents of the corresponding task description file, the file will immediately be written to disk to protect it in case of system failure.

There may be multiple subtasks for each task, and these may be assigned to different KWs.

The data file associated with this object is an ASCII file with one field on each line. These fields are in human readable form. This allows a knowledgeable user to change these files with a text editor, and provides flexibility which should prove useful in rapidly evaluating changes to the prototype.
Attachments to a task are global rather than associated with a particular subtask.

In the production version the file structures can, and probably will, be changed with relatively little impact on the Knowledge Manager code. Only a few of the lowest-level message handlers will have to be recoded to handle the new file formats.

Fields

title

A short title for this task.

description

A more detailed description of the task.

date_started
date_completed

The current status of this task.

date_due

The date this task is due.

hours_needed

Number of hours needed to accomplish this task.

hours_worked

Number of hours worked on this task.

pct_complete

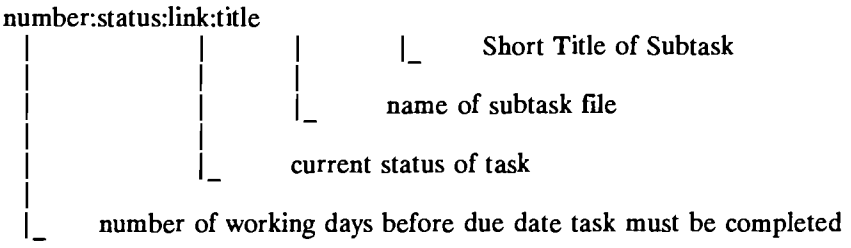
Percent task is complete.

attachments

A list of file names, one per line.

subtask

A list of subtasks associated with this task. There is one subtask per line and the format of the line is:



dependencies

A list of the task ids of tasks which must be completed before this one can begin.

Class Messages

new(self) SUPERVISOR

An empty task with one subtask will be created.

open(self,name) ANY

A task is created and the named file is read into it and displayed.

Instance Messages

selectSubtask(self) ANY

The subtask the user is pointing to becomes the current subtask.

addSubtaskAbove(self,title) SUPERVISOR
addSubtaskBelow(self,title)

A subtask with the specified title is added above or below the currently-selected subtask.

changeSubTask(self,title) KWONLY

The title of the currently selected subtask is changed to the title specified. (The local copy of the title is changed and a message is sent to the subtask to change its title.)

deleteSubtask(self) SUPERVISOR

A message is sent to the currently-selected subtask to delete itself.

selectAttachment(self) ANY

The attachment the user is pointing to becomes the current attachment.

addAttachment(self,type,filename,title) SUPERVISOR

The attachment file is attached globally to this task. It is given the title specified; and a hypertext link of the type specified will be used to link it to this task.

deleteAttachment(self) SUPERVISOR

The currently selected-attachment is deleted.

Subtask

Description

A subtask object is created from the ToDo list window. A subtask object accepts messages from the user and performs the specified actions on a subtask description file. If any message causes a change in the contents of the corresponding subtask description file, the file will immediately be written to disk to protect it in case of system failure.

There may be multiple subtasks for each task, and these may be assigned to different KWs.

The data file associated with this object is an ASCII file with one field on each line. These fields are in human readable form. This allows a knowledgeable user to change these files with a text editor and provides flexibility which should prove useful in rapidly evaluating changes to the prototype.

In the production version the file structures can, and probably will, be changed with relatively little impact on the Knowledge Manager code. Only a few of the lowest-level message handlers will have to be recoded to handle the new file formats.

Fields

assigned_to

Id of the KW to which this subtask is assigned. This field is only editable by a supervisor.

title

A short title for this subtask.

date_started
date_completed

The current status of this task.

date_due

The date this subtask is due.

days_before_task

The number of working days before the task is due that this subtask must be completed.

hours_needed

Number of hours needed to accomplish this task.

hours_worked

Number of hours spent working on task to date.

pct_complete

Percent task is complete.

attachments

A list of file names, one per line.

steps

A list of steps. The format for a step list is: the step title, followed by DoIt list, followed by the attachment list. For example:

```
        Step 1 Title
        D<tab> type link 1 : link 1 : Title link 1
        D<tab> type link 1 : link 1 : Title link 1
                          :
        D<tab> type link 1 : link 1 : Title link 1
        A<tab> type link 1 : link 1 : Title link 1
        A<tab> type link 2 : link 2 : Title link 2
:
        A<tab> type link n : link n : Title link n

    Step 2 Title
                  :
```

Class Messages

new(self) SUPERVISOR

An empty subtask with one step will be created. The id of the person who creates it will be filled into the assigned_to field. This allows that person to continue to change the fields with KWONLY privilege until the subtask is assigned to a KW.

open(self,name) ANY

A subtask is created and the named file is read into it and displayed.

Instance Messages

selectStep(self) ANY

The step the user is pointing to becomes the current step.

addStepAbove(self,title) KWONLY
addStepBelow(self,title)

A step with the specified title is added above or below the currently-selected step.

changeStep(self,title) KWONLY

The title of the currently-selected step is changed to the title specified.

deleteStep(self) KWONLY

The currently-selected step is deleted if it has no attachments. If the step has attachments they must be deleted with deleteAttachment first.

selectAttachment(self) ANY

The attachment the user is pointing to becomes the current attachment.

addAttachment(self,type,filename,title) KWONLY

The attachment file is attached to the currently selected step. It is given the title specified and a hypertext link of the type specified will be used to link it to this task.

deleteAttachment(self) KWONLY

The currently-selected attachment is deleted.

ToDo List

Description

There is only one instance of the ToDo list object. It is created when the Knowledge Manager is started and handles the user messages relating to the subtasks the user must perform.

The ToDo List file contains one line for every task assigned to a knowledge worker. That line consists of the fields, separated by commas. The only required information in this file is the subtask link; the remaining fields are duplicated in this file to reduce file I/O.

The ToDo list for supervisors links to tasks rather than subtasks. This is how supervisors get to the tasks for assigning subtasks and analyzing the schedule.

Fields

duedate

The data the subtask is due. The file is sorted on this field

date_started
date_completed

The current status of this subtask. This is the secondary key for this file.

subtask

The link to the subtask file.

title

The title of this subtask.

pct_complete

The percent complete for this subtask.

Class Messages

new(self)

Issued once by system during start up. Creates a ToDo list.

Instance Messages

close(self) ANY

Closes all other Knowledge Manager windows and exits Knowledge Manager.

selectItem(self) ANY

Sends a open(SubTask,subtask) message using the subtask field of the line pointed to by the user.

addItem(self,duedate,filename) KWONLY

Adds the item to the ToDo list according to the sorted order. Creates a subtask by sending a new message to the subtask.

deleteSelectedItem(self) KWONLY

The currently-selected item is removed from ToDo list and a destroy message is sent to the item itself.

Attachment

Description

An attachment is a file containing any type of data. The only type of data the prototype will be able to display is ASCII text. It is acceptable to attach any other type of data to an object; the Knowledge Manager will keep track of such files.

Fields

NONE

Operations

new(self,filename) ANY

Display the file in a window.

close(self) ANY

Close file and remove window.

Detailed Window Descriptions

ToDo List Window

The ToDo list window is the first window the users sees when the system is started. Each line of the ToDo list contains one subtask. The user can select this task by pointing to it with the mouse and clicking once. If the user points to a line and double clicks the mouse, the associated link is activated and the subtask window appears.

Menu Items

Help

Invoke the Knowledge Manager help function.

Edit

This choice is only available when a ToDo list item is selected. When this choice is selected a submenu will appear with the following choices.

Add

Add a new item to the personal schedule

Delete

Delete the selected item from the personal schedule.

News

Open the news file as an attachment.

Status

Set the reporting status of an item on the list of things to do.

Complete

Mark the item "completed" and remove from list of things to do.

Started

Mark the item "started".

Schedule Window

The schedule window is the entry point to the Knowledge Manager system for users wishing to edit task information. The task due date is directly modifiable from this schedule window. All other task information is editable in the task window which the schedule window allows a user to open.

The user can select a task by pointing to it with the mouse and clicking once. If the user points to a task and double clicks the mouse, the associated link is activated and a task window appears.

Menu Items

Help

Invoke the Knowledge Manager help function.

Edit

Add

Add a new task.

Delete

Delete the currently-selected task.

Modify

Modify the currently-selected task. This is the same function invoked when the user double clicks on a task.

Schedule

WhatIf

Allows the user to change the date field on the currently-selected task. The scheduler will flag all impacted tasks.

Change

Accept the current changes into the master schedule. This button will be grayed for users not authorized to perform this action.

Task Window

The task window allows knowledge workers to view information about a task in general; it also allows users at the supervisor level or above access to change task information.

Menu Items

Help

Invoke the Knowledge Manager help function.

Edit

Add

Add a new subtask to the task.

Delete

Delete the selected subtask from the task.

Modify

Open a subtask window to allow the selected subtask to be modified. This is the same as double clicking on the selected subtask.

Attachments

This dynamic menu contains a list of every attachment connected to the task. At the task level attachments are connected to the task in general, not to an individual subtask.

To access the attachments for a subtask, the user should select "modify" or double click on the subtask, which will open the subtask window.

Assign

Allows the user to edit the assigned_to field of the selected subtask.

Appendix B. -- System Requirements

To run the Knowledge Manager system, a Knowledge Worker must have the following hardware and software:

Software

- MS-DOS 3.1 (or newer).
- Microsoft Windows/386.

Hardware

- An IBM compatible personal computer based on the intel 386 micro-processor.
- 4 megabytes of random access extended memory.
- 60 megabytes of hard disk storage.
- A mouse.
- An EGA or VGA monitor.

KNOWLEDGE WORKER SYSTEM TECHNICAL GUIDE

Prepared Under Contract
DACA88-88-D-0020/0008

Task 3-2
KWS Technical Guide

Submitted to:

Department of the Army
CONSTRUCTION ENGINEERING RESEARCH LABORATORY
Champaign, Illinois

Prepared January 12, 1990 by:
GEORGIA INSTITUTE OF TECHNOLOGY
Atlanta, Georgia

Contents

Overview

This manual describes how the Knowledge Worker System (KWS) prototype software works. The KWS prototype was written in the ACTOR language and runs as a Microsoft Windows application.

Intended Audience

Before reading this manual you should be familiar with ACTOR, Microsoft Windows, and Microsoft DOS operating system.

Familiarity with ACTOR can be gained by reading the Actor Language Manual by the Whitewater Group. This book is included with the Actor Development System. Since the software for the KWS prototype is written in ACTOR, it is extremely important that you understand this material before attempting to modify or extend the KWS software.

KWS is a Microsoft Windows application. This means that the KWS software calls Windows services to perform many of its tasks. While it is possible to get a general idea of how the software works without a detailed knowledge of MS Windows, it would be very difficult to make programming changes to the KWS software without a detailed understanding of the operation of MS Windows. At the very least you should read the "Microsoft Windows User Guide" which comes with the run time version of MS Windows. If you want to make programming changes to the KWS software, you should also familiarize yourself with the manuals in the Microsoft Software Development Kit.

Some routines in the KWS software make direct calls to DOS system services. These calls are documented in the DOS Technical Reference Manual.

Organization of this manual

This manual is divided into four sections, each of which describes a piece of the KWS software. These sections describe; the data organization, the user interface, the file system, and the scheduler.

The section on data organization gives an overview of how the KWS data is manipulated by the KWS system.

The user interface consists of all the code that generates the windows which display user information and accepts command from the user.

The KWS file system is the interface between the KWS system and the DOS file system. It handles moving KWS data in and out of the KWS system.

The scheduler is an implementation of the critical path method (CPM) scheduling algorithm. The scheduler handler is invoked when a user wants to manipulate date-dependent KWS data.

Data Organization

There are two major types of data the KWS prototype manipulates: 1) date-related information (schedule information) and, 2) information not related to a date (usually text).

The date-related information which KWS manipulates corresponds to information about when a subtask is due. These dates are further organized into a hierarchy that groups subtasks into tasks and tasks into events.

The information not related to a date which KWS manipulates consists of lists of steps required to complete some subtask and supporting documentation attached to each of these steps.

These two types of information are logically related to each other by means of hypertext links. A hypertext link is simply the DOS name of a file which contains the related information. For example, a subtask is a date-related item (it has a due date and a short subtask description); it is also related to a series of steps; therefore, a third quantity is used in the description of a subtask, the name of the file which contains the list of steps.

Simple Data Classes

There are three base classes for KWS data. These are Date, HyperLink, and Step.

1 Basic Data Classes

The Date class creates and manipulates date objects. A date object is represented internally as a long int. The date class can respond to: messages to convert the date to various string formats; messages to tell the day of the week a date object refers to; messages to increment and decrement the date; messages to compare two dates in order to determine which one is before the other; or messages to determine the number of days between any two dates.

The HyperLink class is the base class for linking any KWS data object to any other KWS data object. A HyperLink object has a title, a link, and a type. The title is a string and is used when the object is displayed. The link is the name of a DOS file which contains another object. KWS does not use objects in the HyperLink class directly; rather it uses objects which are in the descendant classes: AtchLink, DoitLink, SubTask, Task.

All HyperLink objects accept the message openLink. When the message openLink is received by a HyperLink object or a descendant of a HyperLink object, the object creates a window of the appropriate type to view the

data in its link file and sends this window a message to display the link file.

Attachments are added to KWS by creating a new object in the AtchLink class. Objects in the AtchLink class are the same as HyperLink objects except that their links are assumed to point to files containing ascii text. Additionally, AtchLink objects have a method to convert themselves to DoitLink objects.

In response to an openLink message, an AtchLink object will usually create an AtchWindow. The exception to this rule occurs when the object's type variable indicates the file pointed to in the link field contains a step list; in this case a StepWindow is opened instead.

DoIts are added to KWS by creating a new object in the DoitLink class. DoitLinks, like AtchLinks, are descended from the HyperLink class. Objects in the DoitLink class have links that point to files containing data. Additionally, DoitLink objects have one extra instance variable (program). If program is true, then the data pointed to by the link variable is assumed to be a program to be executed, otherwise the data in the linked file is treated exactly the same as an attachment. DoitLink objects can convert themselves into AtchLink objects.

When a DoitLink object receives a openLink message, it will determine whether or not the file named by the link is executable or not. If the file is not executable, an AtchWindow is created. This version does not yet support automatic execution, therefore DoitLink objects ignore openLink messages to programs.

SubTask objects contain links to step lists. When a subtask object receives an openLink message, it will open a StepWindow. Additionally, SubTask objects are date related. A SubTask object can compare itself chronologically with other SubTask objects, as well as determine whether or not it is currently finished, late, or not due yet. SubTask objects can be sent messages asking them what color they are; they will return a RGB color, based upon whether or not they are finished, late, or on schedule.

The Task class is a descendent of the SubTask class. Therefore, it actually contains information for one subtask. In addition to all the properties of a SubTask, a Task object has some additional properties related to dynamic scheduling. A Task object knows to which Knowledge Worker the subtask it contains is assigned. It also knows the name of the task of which it is a part. A Task object keeps a list of predecessors and successors of itself, and knows its estimated duration.

Composite Data Classes

The KWS data types described above can be aggregated into larger groups, for instance, a group of attachments can be associated with one step or a group of subtask can be associated with one task. The KWS software uses a set of classes to represent this grouping. Aggregate objects in these

classes can respond to messages which apply to all the simple objects as a group.

The base class for aggregating HyperLinks or descendants of HyperLinks is the LinkCollection. This class is a descendant of the ACTOR SortedCollection class. This means any object added to this class must handle the "<" message. The HyperLink class will provide a method to handle this message for all of its descendants that do not provide this method. The default method returns nil (False) for every object; this will always place the object added last to a LinkCollection at the end of the collection. The date- related classes (SubTask and Task) implement the "<" method to insert items into the LinkCollection in chronological order with all finished tasks before any unfinished tasks.

Descended from the LinkCollection is the DoitCollection. This class adds the ability to insert a string containing the printed form of a DoitLink into the collection in internal format. This class is mainly used to keep collections of DoitLinks separate from collections of AtchLinks (which are aggregated as a LinkCollection) with the hope that this will be useful when automatic execution is actually implemented.

The SubtaskCollection, also descended from the LinkCollection, contains methods to convert the printed form of a subtask (usually read from a file) and create a new SubTask object which is then added to the collection. Additionally, objects in this class can add a string containing the printed form of a subtask and add it to a specific place in the collection.

The TaskCollection is derived from the SubTask collection. It is very similar to the SubtaskCollection but aggregates Task objects rather than SubTask objects. The TaskCollection also adds the capability to update the successor lists of each object in it based on their predecessor lists. This is used by the dynamic scheduler.

The Step class is the simplest of the aggregate objects. Objects in this class correspond to one step in the list of steps required to do one subtask. A step object has a title (which contains the description the KW sees) ,a collection of DoIts (in a DoitCollection) and a collection of attachments (in a LinkCollection).

Objects in the Step class are aggregated into objects in the StepCollection class. This class also contains methods for converting objects in this classes to text (for storing in a file), as well a converting text (from a file) into a step collection.

User Interface

The user interface provides the KW with a graphical way to manipulate the data maintained by the KWS program. The two basic types of windows presented to the KW. One window type allows the user to manipulate data as free-form text, the other window type allows the user to traverse the hyperlinks.

Free-Form Text Windows

There are three types of windows that allow the user to manipulate their contents in a free-form manner. These are the NoteWindow, the AtchWindow, and the InfoWindow. Each type of window is created as an object in the NoteWindow, AtchWindow, and InfoWindow class respectively. The InfoWindow is a descendant of the AtchWindow which, in turn, is a descendant of the NoteWindow.

All text edit windows are created and displayed as the result of some user action on a hyperlink window. A NoteWindow object is created and told to display itself when the user selects the Notes button from the ToDo List menu bar. Its window is initially editable. An AtchWindow object is created and told to display itself when the user opens an attachment or non-executable DoIt (either by double clicking in a step window or selecting from the Attachments window). An InfoWindow results from the user selecting the Tips button. Neither AtchWindows or InfoWindows are editable when they are initially displayed.

The NoteWindow class is a descendant of the ACTOR FileWindow class. This class was created by subtracting the properties of the FileWindow class which were not useful. Particularly, the ability to change the name of the file being edited was removed. This restriction was enacted to preserve system integrity. None of the text editing windows can be reached by a user without taking some action in a hyperlink window. Thus, the window is opened as the result of traversing a hyperlink; if the user changed the file name the window which opened the edit window would not be informed and the integrity of the link would be destroyed.

Any object which is in the NoteWindow class or any class descended from NoteWindow also keeps track of how long the user keeps the window open. When one of these objects is created, it is given the name of the subtask to which it is ultimately linked. Each of these objects starts a timer when it gets the input focus; when the object loses input focus, it sends a message to the ToDo list object containing the name of the subtask and how long the window was open, which is displayed as "Time Spent."

Hyperlink Windows

All other windows a user sees when using the KWS prototype are Hyperlink Windows. These windows only allow the user to enter information in a formatted manner by means of dialogue boxes. This is done so the software

can understand and maintain the relationship between various data items (i.e., tasks <-> steps).

Internally each window keeps all the data objects in one aggregate called workText. This aggregate is one of the aggregate types discussed in the previous section. The information in a hyperlink windows is formatted one object to a line, for example, a StepWindow has one step per line; the ToDo window has one subtask to a line.

The HyperWindow class is the base class for all windows of this type. It is a descendant of the ACTOR EditWindow class with the ability to type directly into the window disabled. Additionally, this class redefines the wndIcon class method to load the Castle icon for all windows are in this class or are descended from it.

Descended from the HyperWindow class are classes which add the properties unique to the type of window KWS displays for different types of data. Objects in the class StepWindow control windows that show step lists. Objects derived from TodoWindow control windows that show time-related data: ToDo lists, Event lists, Task lists, etc. The following discussion will refer to any window object in or derived from the HyperWindow class as a hyperwindow and WindowClass shall generically refer to the class under discussion.

Creating a Hyperwindow

A hyperwindow is created by sending an open message to the class as follows:

```
open(WindowClass,"Window Title","filename","c:\path")
```

This results in the following:

open creates a new hyperwindow object and sets the file and path information in it.

Then open searches the KwWindows (a global variable containing a list of all currently open KWS windows) to see if the file is already open.

If the file is open then the new hyperwindow object is destroyed and a message is sent to the window that contains the file and tells it to display itself to the user.

Otherwise, the new hyperwindow is added to KwWindows and its instance variables are initialized by sending it an initWorkText message.

Next, a demoSelf message is sent to the newly-created object.

Finally, the new object is returned.

The method initWorkText initializes any instance variables used by the new hyperwindow.

The method demoSelf calls readText to read the windows data. Next, demoSelf sends a show message to the new hyperwindow to make the object's window visible.

The message readText causes an object to read the data file associated with the object. The file that is read is a text file; the readText method converts it into the type of collection appropriate for the internal representation of the data. For example, in a StepWindow object the readText method will convert the data into a StepCollection.

Displaying Data

The contents of the client area of the hyperwindow is displayed by the object's paint method. The paint method sends the message windowPrint to each object in workText which is in the visible area of the window. The object will return a string to be displayed in the window as a result of the windowPrint message. This string is adjusted for any horizontal scrolling and then displayed in the window.

User Selections

When the user double clicks the mouse while pointing to a line in the client area of a window, Windows sends a WM_LBUTTONDBLCLK message to the window. This message is converted in the base class HyperWindow into a linkTo message. The linkTo method sends an openLink message to the object in workText corresponding to the line pointed to by the user.

Menu Operations

A menu bar is associated with every hyperwindow. At the minimum, it contains a "Help" button and a "FixIt!" button. Some classes of hyperwindows add other menu options specific to the type of information the window contains. There are three messages sent to a hyperwindow to allow it to customize its menu bar.

A loadMenu message is sent to a hyperwindow when it is created to allow it to load a custom menu bar. If loadMenu returns false (nil), the default menu is loaded.

Whenever a hyperwindow gets the input focus (the ability to accept keyboard or mouse input from the user), a updateMenu message is also sent to it. This message allows the window to make customized changes to its menu bar based on events that may have occurred when the user was using a different window.

Finally, whenever a user selects a menu choice that causes a submenu to appear, a WM_INITMENUPOPUP message is sent to the window before the submenu is displayed. This message allows the window to dynamically change the

contents of the submenu. A StepWindow uses this to change create the Attachments and DoIts submenus based on the step the user has selected.

Adding, Deleting, Modifying Data

When the user selects add, delete, or modify from the FixIt menu, the hyperwindow receives an addItem, deleteItem, or modifyItem message respectively. In response to this message, the hyperwindow creates a new dialogue box object containing a copy of the user-selected object.

The dialogue box is then sent the runModal message. When the dialogue box receives this message, it makes itself visible and prevents input to the window that created it until it is closed.

The hyperwindow waits for the dialogue box to close. When the dialogue box is closed, the hyperwindow interrogates the dialogue box to see if the user accepted the change. If the user did accept the change then workText modified to reflect the edit. Finally, the window sends itself a message (invalidateRect) telling itself to redisplay the modified workText.

HyperWindow Descendant Classes

Objects in the StepWindow class manipulate windows containing step lists. The instance variable workText is a StepCollection.

The rest of the hyperwindow classes are descendants of the TodoWindow. The workText variable for these classes of windows is a SubtaskCollection (or its descendant TaskCollection). TodoWindow is the base class for these time-related windows but no KWS window is an instance of this class. The KWS windows derived from this class are:

EventLWindow - These windows are used to display an event list. The workText variable for this type window is a SubtaskCollection. The openLink method in this class causes a TaskLWindow to open.

TaskLWindow - These windows are used to display a task list. The workText variable for this type window is a SubtaskCollection. The openLink method in this class causes a TaskWindow to open.

KWTodoWindow - This is the class from which the ToDo window is created. The workText variable for this type window is a SubtaskCollection. The openLink method in this class opens a StepWindow.

TaskWindow - Windows in this class display the list of subtasks related to one task. This is the only window where all schedule information for a subtask can be entered. The workText of this windows is a TaskCollection. The openLink method in this class opens a StepWindow.

SchedWindow - The SchedWindow class is descended from the TaskWindow class. It has all the properties of the TaskWindow with one exception; this type of window can not read or write to a file directly.

A window of this type is created when the user chooses the Schedule option from the FixIt! menu in either a EventLWindow or a TaskLWindow. A SchedWindow is created with a view message rather than an open message and the workText variable is passed to it. The workText variable type is a Schedule.

File System

The date file Input/Output is all done by objects in the class KmFile. This class is a descendant of the ACTOR class TextFile. The major difference between a KmFile object and a TextFile object is that when an open for reading message is sent to a KmFile object whose file name points to a file which does not exist, it will not fail the operation (as a TextFile object will) but instead will create the file.

Also, the KmFile class has additional methods to allow it to create directories, test file attributes, and apply ACTOR operations to a set of files specified with wild-card characters.

The KWS data files are stored on the hard disk in directories under one of two paths. The global variable KwPrivPath is a string containing the path to the KW's private data. This private data is the information related to items the KW entered through the ToDo list. The global variable KwDataPath is a string containing the DOS path to the public KWS data. This data is related to items entered into the system through the TaskWindow.

The following directories exist under these paths:

Todo - This directory contains the knowledge worker's todo list files. The file names are of the format TODOkwid.TDO.

Schedule - This directory contains the scheduler related files. The file names are formatted as follows:

EVENTS.EVT - contains the list of all events known to KWS.

eventid.EVT - contains the list of all tasks associated with an event.

taskid.tsk - contains the list of all subtasks associated with a task.

Steps - This directory contains on directory for every subtask in the system. Within the subtask directory there will be one file named subtaskid.STP. This file contains the list of steps for performing the subtask. Additionally, the subtask directory is the default directory for all attachments to the steps in the step list file.

Help - This directory contains the system help files.

Scheduler

The KWS prototype incorporates a critical path method (CPM) scheduler. The methods related to scheduling are contained in the class Schedule.

The scheduling algorithms in the KWS prototype schedule all data in one event at a time. It is assumed that events decompose such that one event is completely finished before another one begins. Tasks are not assumed to be independent.

When the user chooses to run the scheduler from either a TaskLWindow or an EventLWindow, a schedule object is created and all the subtasks for the event specified (or the event of which the specified task is part) is read into memory. This schedule object is then told to show itself.

When the user changes the due date of a subtask so that it falls on an earlier date (compresses the due date), the scheduler will propagate this change backward to all the subtasks predecessors. No date can be moved forward earlier than the current day. In the case where a subtask receives a message that requires the subtask's due date be changed to a date earlier than the current day, the due date is set to the current day, the subtask is marked late and shows 0 days slack time.

Tasks are compressed by compressing the due date of each subtask which is part of a task. Events, similarly, are compressed by compressing the due date of all tasks which comprise the event.

When the user changes the due date of a subtask so that it moves into the future (decompresses the due date), the schedule propagates this change to all the successor tasks of the subtask. No subtask will be scheduled so that its due date falls after the due date of the event of which it is part.

Likewise tasks and events are decompressed by decompressing each of their constituent parts.

Appendix I - List of KWS Classes

```
Date

EditAtch
  EditDoit
EditItem
EditStep
EditTodo
  EditTask
  ViewTodo

HelpWindow

HyperLink
  AtchLink
  DoitLink
  SubTask
    Task

HyperWindow
  StepWindow
  TodoWindow
    EventLWindow
      TaskLWindow
    KWTodoWindow
    TaskWindow
      SchedWindow

KmFile

LinkCollection
  DoitCollection
  SubtaskCollection
    TaskCollection

Login

NoteWindow
  AtchWindow
    InfoWindow

Printer
  TextPrinter

Schedule

Step

StepCollection
```

Appendix II - KWS Class Definitions

AtchLink

Source file: ATCHLINK.CLS

Inherits from: Object HyperLink

Inherited by: (no descendants)

class comment

Instance variables:

type (From class HyperLink)
file (From class HyperLink)
title (From class HyperLink)

Class methods: (none)

Object methods:

asDoit(self)

convert attachment object to a doit object.

printOn(self,aStrm)

This method prints an attachment object. This is the format it will appear in a KW file

AtchWindow

Source file: ATCHWIND.CLS

Inherits from: Object WindowsObject Window TextWindow EditWindow WorkEdit FileWindow NoteWindow

Inherited by: InfoWindow

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
isMain (From class Window)
caption (From class Window)
locRect (From class Window)
buttonDn (From class Window)
hMenu (From class Window)
paintStruct (From class Window)
caretVis (From class TextWindow)
chStr (From class TextWindow)
textMetrics (From class TextWindow)
xMax (From class TextWindow)
yPos (From class TextWindow)
xPos (From class TextWindow)
tmHeight (From class TextWindow)
tmWidth (From class TextWindow)
workText (From class TextWindow)
vCol (From class EditWindow)
aCol (From class EditWindow)
dragDir (From class EditWindow)
endLine (From class EditWindow)
endChar (From class EditWindow)
startLine (From class EditWindow)
startChar (From class EditWindow)
dirty (From class EditWindow)
leftChar (From class EditWindow)
topLine (From class EditWindow)
dragLine (From class EditWindow)
oldX (From class EditWindow)
pOrigin (From class EditWindow)
dragDC (From class EditWindow)
iD (From class WorkEdit)
outputDelimiter (From class FileWindow)
inputDelimiter (From class FileWindow)

file (From class FileWindow)
fileDlg (From class FileWindow)
task (From class NoteWindow)
time (From class NoteWindow)
curPath (From class NoteWindow)
canEdit (From class NoteWindow)

Class methods: (none)

Object methods:

fileName(self)

This method returns the defualt file name for a file associated with the window.

init(self)

Initialize local variables, Initially NoteWindows has editing turned off

Date

Source file: DATE.CLS

Inherits from: Object Magnitude

Inherited by: (no descendants)

Object to hold a date -- created 8/26/88 acr

11/30/88 mzu -added a test in asString to use international date format. -asLongString, dayOfWeek are hard coded in english

Instance variables:

year Year
day Day 1-31
month Month 1-12

Class methods:

current(self)

Return current date.

new(self)

Create a new Date object and initialize.

Object methods:

+(self, x)

Infix operation. Add a Date to another Date, an Int, or a Long. Return a Long. You must use asDate() to convert back to a Date.

-(self, x)

Infix operation. Find difference between two dates, or subtract a number of days (an Int or Long) from a Date.

<(self, x)

Infix operation. Return 0 if x < self, otherwise return nil.

<=(self, x)

Infix operation. Return 0 if x <= self, otherwise return nil.

=(self, x)

Infix operation. Return 0 if x = self, otherwise returns nil.

>(self, x)

Infix operation. Return 0 if x > self, otherwise return nil.

>=(self, x)

Infix operation. Return 0 if x >= self, otherwise return nil.

asDDMMYYString(self)

Return date in abbreviated DD/MM/YY string format.

asDayString(self)

Return date in 'day of the week' format.

asInt(self)

Included so that addition and subtraction will work with Longs or Ints.

asLong(self)

Return number of days since 1/1/1901.

asLongString(self)

Return date in long string format.

asMMDDYYString(self)

Return date in abbreviated MM/DD/YY string format.

asMilString(self)

Return date in long string format.

asString(self)

Return date in string in international date format

asYYMMDDString(self)

Return date in abbreviated YY/MM/DD string format.

dayOfWeek(self)

Return day of the week.

daysBeforeMonth(self)

Private method. Return number of days in a year before this month.

dec(self)

Decrement a Date by one, modifying it directly.

diff(self, y)

Return number of days between two dates. This is a more efficient than date1 - date2.

inc(self)

Increment a Date by one, modifying it directly.

init(self)

Initialize a date to 1/1/80.

isValid(self)

Determine if a Date is valid. Used by conversion methods to guarantee that dates are always valid.

leapDays(self)

Return number of February 29s since 1/1/1901.

next(self)

Given a date, return the next date. Preserve original date.

previous(self)

Given a date, return the previous date. Preserve original date.

print(self)

Print a date.

printOn(self, aStrm)

Put a date onto the specified Stream.

DoitCollection

Source file: DOITCOLL.CLS

Inherits from: Object Collection IndexedCollection Array
OrderedCollection SortedCollection LinkCollection

Inherited by: (no descendants)

This class contains the methods used to manipulate a collection of doit links

Instance variables:

lastElement (From class OrderedCollection)
firstElement (From class OrderedCollection)
compareBlock (From class SortedCollection)

Class methods: (none)

Object methods:

insertLinkStr(self, aStr, idx)

Inserts the string format of a DoItlink into the collection as a link in internal format

DoitLink

Source file: DOITLINK.CLS

Inherits from: Object HyperLink

Inherited by: (no descendants)

class comment

Instance variables:

type (From class HyperLink)
file (From class HyperLink)
title (From class HyperLink)
program program to execute link

Class methods: (none)

Object methods:

asAtch(self)

This method converts a doit object to an attachment object.

openLink(self,fdir)

This method simulates the execution of a doit link if the DoIt is executable then the a fake result is created, otherwise the a window is opened containing the contents of the associated file.

printOn(self,aStrm)

This method prints an DoIt object. This is the format it will appear in a KW file

program(self)

This method returns true if the DoIt is executable.

setLink(self,aStr)

This method translates a string containing printed format into internal format.

toggleType(self)

This method marks executable DoIts as nonexecutable, and vice-versa.

EditAtch

Source file: EDITATCH.CLS

Inherits from: Object WindowsObject Dialog FileDialog

Inherited by: EditDoit

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
editFocus (From class FileDialog)
pathSpec (From class FileDialog)
fileSpec (From class FileDialog)
loadFile (From class FileDialog)
startDir (From class FileDialog)
defaultDir
attachment

Class methods:

new(self,anAtch, file,dflt)

Create a dialog to edit a todo line

Object methods:

command(self, wP, lP)

Handle file dialog events (OK, Cancel, etc.). Selecting Accept (IDOK) will always do something, if only to reload the list according to the current filter.

getRelFile(self)

This method returns a relative path to the file name for the attachment unless the file is not in the default directory for attachments, in which case the absolute file name is returned. In either case if the attachment is marked as a link to a step the file extension is stripped.

initDialog(self, wp, lp)

Initialize the file loader dialog.

EditDoit

Source file: EDITDOIT.CLS

Inherits from: Object WindowsObject Dialog FileDialog EditAtch

Inherited by: (no descendants)

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
editFocus (From class FileDialog)
pathSpec (From class FileDialog)
fileSpec (From class FileDialog)
loadFile (From class FileDialog)
startDir (From class FileDialog)
defaultDir (From class EditAtch)
attachment (From class EditAtch)

Class methods: (none)

Object methods:

initDialog(self, wp, lp)

Initialize the file loader dialog.

EditItem

Source file: EDITITEM.CLS

Inherits from: Object WindowsObject Dialog

Inherited by: (no descendants)

This class creates a dialog box which asks for the name of a subtask to be added to the predicessor list of another subtask.

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
response

Class methods: (none)

Object methods:

end(self, val)

End this dialog, passing the value to MS-Windows. This value is then returned via the Call to the DialogBox function.

entry(self)

This method returns what ever the user entered in as the name of the subtask.

initDialog(self, wP, lP)

By returning a 1 from the INITDIALOG message, we are telling MS-Windows to set the input focus to first tabstop item. (See MS-Windows Reference).

setItem(self,aStr)

This method is called before the dialog box is displayed on the screen and allows the program to preset the input field.

EditStep

Source file: EDITSTEP.CLS

Inherits from: Object WindowsObject Dialog

Inherited by: (no descendants)

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
defaultExt
defaultDir
attchment
step

Class methods:

new(self,aStep, dir, ext)

Create a dialog to edit a todo line

Object methods:

addString(self, item, aStr, idx)

Add aStr to the ListBox, maintaining sorted order, and return its index.

command(self, wp, lp)

This method process user input when the dialog box is active

deleteString(self, item,idx)

Add aStr to the ListBox, maintaining sorted order, and return its index.

editAtch(self)

This method allows the user to edit an attachment link. It creates an edit attachment dialog box using the information from the attachment the user has selected.

editDoit(self)

This method allows the user to edit an doit link. It creates an edit doit dialog box using the information from the doit the user has selected.

getSelIdx(self, item)

Return the index of whatever item is currently selected and set selIdx. Return nil if no item is selected.

initDialog(self, wP, lP)

Initialize the dialog. Note items 110 and 111 are the >> and << buttons

moveAtch(self)

This method moves an the currently selected attachment to the doit box

moveDoit(self)

This method moves the currently selected doit to the attachment box

newAtch(self)

This method creates a new attachment and opens an edit attachment dialog box for the user to edit.

newDoit(self)

This method creates a new doit and opens a edit doit dialog box for the user to edit the contents of the attachment.

EditTask

Source file: EDITTASK.CLS

Inherits from: Object WindowsObject Dialog FileDialog EditTodo

Inherited by: (no descendants)

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
editFocus (From class FileDialog)
pathSpec (From class FileDialog)
fileSpec (From class FileDialog)
loadFile (From class FileDialog)
startDir (From class FileDialog)
task (From class EditTodo)

Class methods: (none)

Object methods:

addAft(self)

This method handles the adding of a successor to a task. It opens an EDIT_ITEM dialog, and if the user choses accept, adds the file name supplied to the successor list of a task.

addString(self, item, aStr)

Add aStr to the ListBox, maintaining sorted order, and return its index.

addb4(self)

This method handles the adding of a predicessor to a task. It opens an EDIT_ITEM dialog, and if the user choses accept, adds the file name supplied to the predicessor list of a task.

command(self, wp, lp)

Exit point for a simple dialog. If Cancel was not chosen, wP is passed to the MS-Windows EndDialog routine. If Cancel was chosen, 0 is passed to EndDialog. The value passed is then returned by the MS-Windows DialogBox function.

end(self, val)

End this dialog, passing the value to MS-Windows. This value is then returned via the Call to the DialogBox function.

fillList(self,anLB,aCol)

This method adds each line in the collection aCol to the listbox specified, The list box is cleared first.

initDialog(self, wP, lP)

Initialize the dialog.

modAft(self)

This method handles modifying or deleteing an item in the succesor list. It opens and EDIT_ITEM dialog box with the name of the selected link. If the delete button is pressed the link is removed from the list, if accept is chosen the changes overwrite the currently selected link. If cancel chosen nothing is changed.

modb4(self)

This method handles modifying or deleteing an item in the predicessor list. It opens and EDIT_ITEM dialog box with the name of the selected link. If the delete button is pressed the link is removed from the list, if accept is chosen the changes overwrite the currently selected link. If cancel chosen nothing is changed.

EditTodo

Source file: EDITTODO.CLS

Inherits from: Object WindowsObject Dialog FileDialog

Inherited by: EditTask ViewTodo

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
editFocus (From class FileDialog)
pathSpec (From class FileDialog)
fileSpec (From class FileDialog)
loadFile (From class FileDialog)
startDir (From class FileDialog)
task

Class methods:

new(self,aSubTask)

Create a dialog to edit a todo line

Object methods:

command(self, wp, lp)

Exit point for a simple dialog. If Cancel was not chosen, wP is passed to the MS-Windows EndDialog routine. If Cancel was chosen, 0 is passed to EndDialog. The value passed is then returned by the MS-Windows DialogBox function.

end(self, val)

End this dialog, passing the value to MS-Windows. This value is then returned via the Call to the DialogBox function.

initDialog(self, wP, lP)

Initialize the dialog.

EventLWindow

Source file: EVENTLWI.CLS

Inherits from: Object WindowsObject Window TextWindow EditWindow HyperWindow TodoWindow

Inherited by: TaskLWindow

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
isMain (From class Window)
caption (From class Window)
locRect (From class Window)
buttonDn (From class Window)
hMenu (From class Window)
paintStruct (From class Window)
caretVis (From class TextWindow)
chStr (From class TextWindow)
textMetrics (From class TextWindow)
xMax (From class TextWindow)
yPos (From class TextWindow)
xPos (From class TextWindow)
tmHeight (From class TextWindow)
tmWidth (From class TextWindow)
workText (From class TextWindow)
vCol (From class EditWindow)
aCol (From class EditWindow)
dragDir (From class EditWindow)
endLine (From class EditWindow)
endChar (From class EditWindow)
startLine (From class EditWindow)
startChar (From class EditWindow)
dirty (From class EditWindow)
leftChar (From class EditWindow)
topLine (From class EditWindow)
dragLine (From class EditWindow)
oldX (From class EditWindow)
pOrigin (From class EditWindow)
dragDC (From class EditWindow)
aMenu (From class HyperWindow)
fileStamp (From class HyperWindow)
path (From class HyperWindow)

file (From class HyperWindow)
outputDelimiter (From class HyperWindow)
inputDelimiter (From class HyperWindow)
gettingFocus (From class HyperWindow)
hLinkMenu (From class HyperWindow)
hyperLinks (From class HyperWindow)
times (From class TodoWindow)
hWMSize (From class TodoWindow)
hWindowMenu (From class TodoWindow)
todo (From class TodoWindow)
updates
hasChild

Class methods: (none)

Object methods:

applyUpdates(self)

This method is called when an Event list window is openned. It enters the update information (previously read in by readUpdates) to the schedule data. The schedule is read in event by event to keep the memory requirements down.

fileName(self)

This method returns the defualt file name for a file associated with the window.

help(self)

Open the default help window for this window class.

linkTo(self, aLink)

This method opens a window for the link passed in aLink. It also passes the new window a handle to itself used in tracking the time spent in the new window.

loadMenu(self, menuName)

Load the menu resource if possible and obtain a handle to a menu to place in hMenu (if menuName not nil).

makeTodo(self)

This method generates the todo lists for each KW. It sequentially reads the schedule event by event, and assigning the subtasks after each event.

readLinks(self)

Replace the contents of the current window with the contents of its associated data file. The file is read one line at a time and then inserted into workText. Finally, a convert message is sent to workText to handle any class specific initialization that may be necessary. The event list window also processes any update files from the KWs at this time.

readUpdates(self)

This method is called when an Event list window is openned. It reads in all the files containing information about the status changes. There is one file for each KW on the system. The files are deleted after they have been read.

scheduleItem(self)

This method starts the dynamic scheduler. The user is presented with a dialog box which allows editing of the event fields. If the user changes the due date then that changes is applied to every subtask in the event. Finally a schedule windows is openned.

updFinish(self,fl)

This method is called repeatedly for each update file in the system. It reads the update file adding the contents to the updates variable.

HelpWindow

Source file: HELPWIND.CLS

Inherits from: Object WindowsObject Window TextWindow EditWindow

Inherited by: (no descendants)

The methods in this class open a help window and display a help file. A help file is simple an ascii file containing the help text.

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
isMain (From class Window)
caption (From class Window)
locRect (From class Window)
buttonDn (From class Window)
hMenu (From class Window)
paintStruct (From class Window)
caretVis (From class TextWindow)
chStr (From class TextWindow)
textMetrics (From class TextWindow)
xMax (From class TextWindow)
yPos (From class TextWindow)
xPos (From class TextWindow)
tmHeight (From class TextWindow)
tmWidth (From class TextWindow)
workText (From class TextWindow)
vCol (From class EditWindow)
aCol (From class EditWindow)
dragDir (From class EditWindow)
endLine (From class EditWindow)
endChar (From class EditWindow)
startLine (From class EditWindow)
startChar (From class EditWindow)
dirty (From class EditWindow)
leftChar (From class EditWindow)
topLine (From class EditWindow)
dragLine (From class EditWindow)
oldX (From class EditWindow)
pOrigin (From class EditWindow)
dragDC (From class EditWindow)

Class methods: (none)

Object methods:

create(self, par, wName, rect, style)

Modify the style for edit windows--add a vertical scroll bar and horizontal scroll bar. Permits edit windows to be tile, popup, or child style.

getHelp(self,file)

Load the specified help file into the window. The file is assumed to be in the HELP directory under the KW data path.

HyperLink

Source file: HYPERLIN.CLS

Inherits from: Object

Inherited by: AtchLink DoitLink SubTask Task

Objects in this class are the basic hyper link. The have a title and are associated with a file. The type of the object describes whether they are programs.

Instance variables:

type type of link
file file that is object of link
title name to be given to link

Class methods: (none)

Object methods:

<(self,aLink)

When HyperLink objects are collected they are often collected in a sorted collection. The sorting order for a hyperlink is put the next one at the end of the list. This method is used to maintain compatibility with subtask links which are sorted by finished and date.

file(self)

This method returns the file name associated with the object.

isStep(self)

This method returns true if the object is a step.

openLink(self,fdir)

This method opens a window which contains the information in the file pointed to by the link associated with this object.

printOn(self,aStrm)

This method prints the object on the stream specified.

setFile(self,afile)

This method associates the file specified by afile with the object.

setLink(self ,aStr)

Translate a string containing printed format into internal format

setTitle(self,afile)

This method associates the title passed in afile with the object.

title(self)

This method returns the title associated with the object.

toggleType(self)

This method toggles the link type associated with the object. If the link type is not a link to a step then it becomes a link to a step, or vice-versa.

windowPrint(self)

this method returns as a string the information a user would see in a window

HyperWindow

Source file: HYPERWIN.CLS

Inherits from: Object WindowsObject Window TextWindow EditWindow

Inherited by: StepWindow TodoWindow EventLWindow TaskLWindow
KWTodoWindow TaskWindow SchedWindow

HyperWindow

The base class for hyper text windows.

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
isMain (From class Window)
caption (From class Window)
locRect (From class Window)
buttonDn (From class Window)
hMenu (From class Window)
paintStruct (From class Window)
caretVis (From class TextWindow)
chStr (From class TextWindow)
textMetrics (From class TextWindow)
xMax (From class TextWindow)
yPos (From class TextWindow)
xPos (From class TextWindow)
tmHeight (From class TextWindow)
tmWidth (From class TextWindow)
workText (From class TextWindow)
vCol (From class EditWindow)
aCol (From class EditWindow)
dragDir (From class EditWindow)
endLine (From class EditWindow)
endChar (From class EditWindow)
startLine (From class EditWindow)
startChar (From class EditWindow)
dirty (From class EditWindow)
leftChar (From class EditWindow)
topLine (From class EditWindow)
dragLine (From class EditWindow)
oldX (From class EditWindow)
pOrigin (From class EditWindow)
dragDC (From class EditWindow)
aMenu second menu var

fileStamp time of current file
path the current path
file current file
outputDelimiter current file
inputDelimiter current file
gettingFocus button down because were getting focus
hLinkMenu handle to link menu
hyperLinks collection of hyper text links

Class methods:

newWClass(self, lpCl, lpIcon)

Create a new window class Struct.

open(self,title, fName, fPath)

Create a new hyperwindow

sizeRect(self)

Return a sizing rectangle, based on the number of open windows, ideal for positioning a new PopupWindow.

wndClass(self)

Return static string for this window class name ("EditWindow").

wndIcon(self)

Object methods:

##WM_DESTROY(self, wp, lp)

Self has been removed from the screen. Remove from the set of open windows. If self is the application's main window, terminate the application when window is destroyed.

WM_LBUTTONDBLCLK(self , wp, lp)

Windows message sent when the Left mouse button is double clicked.

WM_MENUSELECT(self,wP,lP)

Windows message sent when a menu item has been selected.

WM_RBUTTONDOWN(self, wp, lp)

MS-Window's message for right button release. Default action is to bring browsers to top.

addItem(self)

Add an item to this window. This method catches the addItem message that no desendant handles, it does nothing.

addLinkMenu(self)

This method add any extra menu choices to the base menu bar. Specifically it add the HELP and FixIt choices.

addPopupItems(self,aMenu,menuSize,choices,itemno)

This method replaces the list of menu choices in the popup menu aMenu with the list supplied in choices. The menu id for these choices starts with the number in itemno and is incremented by one for each subsequent choice in choices. menuSize is the number of items in aMenu before the replacement is made.

beginDrag(self, wp, pt)

Initialize the dragging parameters. De-select any previously selected text. Move caret to new position. Initialize the drag-direction flag, dragDir, to nil.

canDoIt(self,aBool)

This method grays the DoIt choice if aBool==false otherwise it enables the DoIt menu choice.

changeProfile(self)

This method opens the login window in response the "change Profile" menu choice.

charIn(self, wP, lP)

Process MS-Window's character input message. Hyperwindows ignore keyboard input to the client area of the window.

command(self, wP, lP)

Dispatch menu choices, accelerators.

convert(self)

This method is called by readlinks, default case is do nothing.

create(self, par, wName, rect, style)

Force the style for self to be tiled/overlapped.

defaultFdir(self)

This method returns the default file path for the a hyperwindow

deleteItem(self)

Delete the item the user has the selection bar on. This method will open a dialog box and verify that the user really wants to delete the selected item before deleting it.

demoSelf(self)

This method read the data file assocated with the window, add the window specific menu choices and shows the window.

drag(self, wp, pt)

Show selected text while the mouse is dragged. The instance variable dragDir is used as a drag-direction flag: 0 for down, 1 for up, and nil for neither. All Hyperwindows ignore this message.

drawString(self, aStr)

Draw a string in the TextWindow.

endDrag(self , wp, lp)

This method changes the line the user has just single clicked on to reverse video (i.e. show the selector bar). It deselects any previously selected item. StartLine will contain the index number to the selected item.

fileName(self)

This method returns the defualt file name for a file associated with the window.

finished(self)

This method handles the finished message for all hyperwindows execpt the todo windows. It does nothing.

ginitTextColors(self, hdc)

Initialize the text and background colors for printing text. Use the given display context.

gotFocus(self, hWndPrev)

Prepare window for input and output, show selected text.

help(self)

Open the default help window for this window class. This class never actually has an open window on the screen, therefor there is no help for it.

info(self)

This method displays the tips window for the current window.

init(self)

initialize delimiter variables

initEditParms(self)

Initialize the editing parameters (home the caret, etc.).

initWorkText(self)

Initialize the workText instance variable, add one zero-length string.

invSelTxt(self)

Invert the selected text if in view. Assumes a valid dragDC. Check for selected text as a condition to invert.

isSelText(self)

Return true if there is selected text.

linkTo(self, aLink)

This method opens a window for the link passed in aLink. It also passes the new window a handle to itself used in tracking the time spent in the new window.

loadMenu(self,name)

There are no menu resources for a hyperwindow don't bother trying to read one.

makeTodo(self)

Stuff any messages about makeing the todo lists that are not handled by a descendant into the bit bucket.

maxChars(self)

Return the length of the longest string in workText.

maxFileSize(self)

Return the maximum file size available to read.

modifyItem(self)

Catch any unhandled modify item messages.

openAtch(self,num)

Ignore any request to open attachments.

openDoit(self,num)

Ignore any requests to open a DoIt, this is a descendants job.

paint(self, hdc)

Redraw the workText from topLine down, and from leftChar rightwards, preserve xPos and yPos. If window has focus, show selected text.

printFile(self)

Send the contents of the current window to the printer, or display an error message if there is a problem.

reInit(self)

Ignore any reinitialize message not handled by a descendant.

readLinks(self)

Replace the contents of the current window with the contents of its associated data file. The file is read one line at a time and then inserted into workText. Finally, a convert message is sent to workText to handle any class specific initialization that may be necessary.

scheduleItem(self)

Start Scheduling system by openning an Event List window.

setFile(self, fName, apath)

Associate the default file name and default file path with this window.

updateMenu(self)

Handle any messages to update the menu bar for descendants that do not handle them.

writeLinks(self)

Write the contents of the window to the associated data file.

InfoWindow

Source file: INFOWIND.CLS

Inherits from: Object WindowsObject Window TextWindow EditWindow
WorkEdit FileWindow NoteWindow AtchWindow

Inherited by: (no descendants)

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
isMain (From class Window)
caption (From class Window)
locRect (From class Window)
buttonDn (From class Window)
hMenu (From class Window)
paintStruct (From class Window)
caretVis (From class TextWindow)
chStr (From class TextWindow)
textMetrics (From class TextWindow)
xMax (From class TextWindow)
yPos (From class TextWindow)
xPos (From class TextWindow)
tmHeight (From class TextWindow)
tmWidth (From class TextWindow)
workText (From class TextWindow)
vCol (From class EditWindow)
aCol (From class EditWindow)
dragDir (From class EditWindow)
endLine (From class EditWindow)
endChar (From class EditWindow)
startLine (From class EditWindow)
startChar (From class EditWindow)
dirty (From class EditWindow)
leftChar (From class EditWindow)
topLine (From class EditWindow)
dragLine (From class EditWindow)
oldX (From class EditWindow)
pOrigin (From class EditWindow)
dragDC (From class EditWindow)
iD (From class WorkEdit)
outputDelimiter (From class FileWindow)
inputDelimiter (From class FileWindow)

file (From class FileWindow)
fileDlg (From class FileWindow)
task (From class NoteWindow)
time (From class NoteWindow)
curPath (From class NoteWindow)
canEdit (From class NoteWindow)

Class methods: (none)

Object methods:

fileName(self)

This method returns the defualt file name for a file associated with the window.

readText(self,f)

This is a debug routine, it shows the amount of memory left in the system in all info windows, instead of the contents of the associated file.

KWTodoWindow

Source file: KWTODOWI.CLS

Inherits from: Object WindowsObject Window TextWindow EditWindow HyperWindow TodoWindow

Inherited by: (no descendants)

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
isMain (From class Window)
caption (From class Window)
locRect (From class Window)
buttonDn (From class Window)
hMenu (From class Window)
paintStruct (From class Window)
caretVis (From class TextWindow)
chStr (From class TextWindow)
textMetrics (From class TextWindow)
xMax (From class TextWindow)
yPos (From class TextWindow)
xPos (From class TextWindow)
tmHeight (From class TextWindow)
tmWidth (From class TextWindow)
workText (From class TextWindow)
vCol (From class EditWindow)
aCol (From class EditWindow)
dragDir (From class EditWindow)
endLine (From class EditWindow)
endChar (From class EditWindow)
startLine (From class EditWindow)
startChar (From class EditWindow)
dirty (From class EditWindow)
leftChar (From class EditWindow)
topLine (From class EditWindow)
dragLine (From class EditWindow)
oldX (From class EditWindow)
pOrigin (From class EditWindow)
dragDC (From class EditWindow)
aMenu (From class HyperWindow)
fileStamp (From class HyperWindow)
path (From class HyperWindow)

file (From class HyperWindow)
outputDelimiter (From class HyperWindow)
inputDelimiter (From class HyperWindow)
gettingFocus (From class HyperWindow)
hLinkMenu (From class HyperWindow)
hyperLinks (From class HyperWindow)
times (From class TodoWindow)
hWMSize (From class TodoWindow)
hWindowMenu (From class TodoWindow)
todo (From class TodoWindow)

Class methods: (none)

Object methods:

WM_INITMENUPOPUP(self,wp, lp)

This method is called when a menu item is selected by the user which has a popup menu associated with it. It is called before the menu is displayed. On this screen the are two, one is the FixIt menu, the other is the Windows menu. This method enables or grays the choices available to the user on the FixIt menu, and updates the list of currently open KW windows.

WM_RBUTTONDOWN(self, wp, lp)

MS-Window's message for right button release. Default action is to bring Todo window to top.

addItem(self)

Add an item to this window. A dialog box is presented to the user, if accept is chosen from the dialog box then the item is added in the the window according to sorting order. (by finished by date)

gotFocus(self, hWndPrev)

Prepare window for input and output, show selected text. Additionally, record the time spent working on any subtask since last time todo window got the user input focus.

modifyItem(self)

Modify the selected item. Display a dialog box with any editable fields and replace the item with the modifications if the user choses accept.

reInit(self)

Reinitialize the data in the window. This message is sent as a result of the user changing the profile.

readLinks(self)

Replace the contents of the current window with the contents of its associated data file. The file is read one line at a time and then inserted into workText. Finally, a convert message is sent to workText to handle any class specific initialization that may be necessary. A KW's todo list has two parts, the private tasks, and the scheduled tasks. These are read seperately.

readPartTodo(self,file,path,isPrivate)

Replace the selection range with the contents of a file. fName is a string naming the DOS file to be read, e.g. "test.dat".

writeLinks(self)

comment

KmFile

Source file: KMFILE.CLS

Inherits from: Object File TextFile

Inherited by: (no descendants)

This class contains all methods used in manipulating the Knowedge Worker System files.

Instance variables:

fileName (From class File)
handle (From class File)
bufLen (From class TextFile)
delimiter (From class TextFile)
position (From class TextFile)
buffer (From class TextFile)
DTA
filePath

Class methods:

cwd(self)

this method returns the current working directory

exists(self,file)

This method returns true if the specified file exists.

parseFile(self,file)

break a valid file spec into drive, path, file

Object methods:

apply(self,aBlk)

This method applies the block argument to every file matching the the pattern specified in the file name of the object. The file name of the object must have been set previously (setName) and is assumed to contain wildcard characters.

atDTA(self)

This method is used internally to set the file name of the object to the name contained in the dist transfer area. This is a support method for findFirst and findNext.

attr(self , aStr)

This method returns the file attribute bits for the object.

createPath(self, path)

This method will create all subdirectories necessary to make path a valid dos path.

dosError(self, bp, str)

Displays an error dialog box with "File Error" as the caption. The ivar, str, contains the name of the file. (The checkError method for this class uses this method.)

findFirst(self , aStr)

This method calls the dos find first function. If a match is found, the file name is set to the returned name.

findNext(self)

This method is used to return subsequent matches after a findFirst. If a match is found then the object's file name is set to it.

getDTA(self)

This method is used internally to get the current disk transfer area address.

getFileDate(self)

This method returns the datestamp of the file associated with the object. The result is a date object.

getFileTime(self)

This function returns the timestamp of the file associated with the object. The result is a string.

getTime(self)

This method returns the time and date of a file as an array of ints.

isDrive(self,aStr)

this routine returns true if aStr is an absoulte path aStr has had trailing \'s removed

mkdir(self , aStr)

This method creates the directory specified by aStr. If the directory can not be created the attribute returned will not have the directory bit set.

open(self, mode)

This method opens a KWS file. If the file does not exist it will create it. It will also create any directories necessary to open the file.

LinkCollection

Source file: LINKCOLL.CLS

Inherits from: Object Collection IndexedCollection Array OrderedCollection SortedCollection

Inherited by: DoitCollection SubtaskCollection TaskCollection

This class contains methods for manipulating collections of hypertext links.

Instance variables:

lastElement (From class OrderedCollection)
firstElement (From class OrderedCollection)
compareBlock (From class SortedCollection)

Class methods: (none)

Object methods:

asText(self)

Return the object in a form suitable for inserting in a text collection. This is the catch all method for hyperlink conversion, and does nothing.

insertLinkStr(self, aStr, idx)

Inserts the string format of a hyperlink into the collection as a link

insertText(self, aStr, line, pos)

Insert a string of lines delimited by CR_LF into the collection at the specified line and character position. Fixes problem where a CR_LF pair which splits across a binary read (usually an exact multiple of 512 bytes) appears as two boxes in a FileWindow

titles(self)

Return a collection of all the titles in the link collection in the same order they appear in the link collection.

Login

Source file: LOGIN.CLS

Inherits from: Object WindowsObject Dialog

Inherited by: (no descendants)

This classes contains the methods used in processing the screen displayed by the "change profile" option in FixIt.

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
canSched
privPath
kwPath
kwid
name

Class methods: (none)

Object methods:

command(self, wp, lp)

Exit point for a simple dialog. If Cancel was not chosen, wP is passed to the MS-Windows EndDialog routine. If Cancel was chosen, 0 is passed to EndDialog. The value passed is then returned by the MS-Windows DialogBox function.

end(self, val)

This method sets the global variables used by KWS to the values the user supplied in the change profile window.

getCheck(self)

Return the check box or radio button state.

initDialog(self, wP, lP)

This method fills a login dialog box (change profile menu choice) with the global variables currently in effect.

setCheck(self, flag)

Set the check box or radio button state.

NoteWindow

Source file: NOTEWIND.CLS

Inherits from: Object WindowsObject Window TextWindow EditWindow WorkEdit FileWindow

Inherited by: AtchWindow InfoWindow

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
isMain (From class Window)
caption (From class Window)
locRect (From class Window)
buttonDn (From class Window)
hMenu (From class Window)
paintStruct (From class Window)
caretVis (From class TextWindow)
chStr (From class TextWindow)
textMetrics (From class TextWindow)
xMax (From class TextWindow)
yPos (From class TextWindow)
xPos (From class TextWindow)
tmHeight (From class TextWindow)
tmWidth (From class TextWindow)
workText (From class TextWindow)
vCol (From class EditWindow)
aCol (From class EditWindow)
dragDir (From class EditWindow)
endLine (From class EditWindow)
endChar (From class EditWindow)
startLine (From class EditWindow)
startChar (From class EditWindow)
dirty (From class EditWindow)
leftChar (From class EditWindow)
topLine (From class EditWindow)
dragLine (From class EditWindow)
oldX (From class EditWindow)
pOrigin (From class EditWindow)
dragDC (From class EditWindow)
iD (From class WorkEdit)
outputDelimiter (From class FileWindow)
inputDelimiter (From class FileWindow)

file (From class FileWindow)
fileDlg (From class FileWindow)
task
time
curPath
canEdit

Class methods:

open(self,title,aFile, aPath)

comment

sizeRect(self)

Return a sizing rectangle, based on the number of open windows, ideal for positioning a new PopupWindow.

Object methods:

WM_DESTROY(self, wp, lp)

Self has been removed from the screen. Remove from the set of open windows. If self is the application's main window, terminate the application when window is destroyed.

WM_INITMENUPOPUP(self,wp, lp)

Before the FixIt popup menu is displayed, enable or disable editing bottons depending on whether or not the window is editable.

WM_RBUTTONDOWN(self, wp, lp)

MS-Window's message for right button release. Default action is to bring the todo window to top.

charIn(self, wP, lP)

Handle auto-indent.

command(self,wp,lp)

Process user input

fileName(self)

This method returns the defualt file name for a file associated with the window.

gotFocus(self, hWndNew)

Prepare this window for user input. Also record current time for use in calculating how long the user was working on the information in this window.

help(self)

Open the default help window for this window class.

init(self)

Initialize local variables, Initially NoteWindows has editing turned on

isEditable(self)

returns true if the editing is turned on in objects window.

loadMenu(self, menuName)

Load the menu resource if possible and obtain a handle to a menu to place in hMenu (if menuName not nil).

losingFocus(self, hWndNew)

When losing focus, de-select text visually, and then hide and destroy the caret.

print(self)

comment

readText(self, fName)

Replace the selection range with the contents of a file. fName is a string naming the DOS file to be read, e.g. "test.dat".

setPath(self,aPath)

Associate a path with the object in the window.

setTrack(self,file,hPrevWnd)

Associate with this object the name of the subtask that this attchment is associated with.

showTitle(self)

Display the current title of the file being edited or "Untitled" if no name has been specified.

Printer

Source file: PRINTER.CLS

Inherits from: Object

Inherited by: TextPrinter

General Printer Class to output to a printer Mark Solinksi, The Whitewater Group.

Instance variables:

port Printer port
driver driver file for printer
device current printer
docName name of document
hPrintDC handle to the printer display context

Class methods:

new(self)

new send an init to instance of self

Object methods:

abortDoc(self)

Escape ABORTDOC procedure

createDC(self)

create a new device context for the printer

deleteContext(self)

delete the printer device context

draftModeOff(self)

Turn draft mode off

draftModeOn(self)

Turn draft mode on

endDoc(self)

Escape ENDDOC procedure

eof(self)

End of File. Return a nil if this command is unsuccessful

escape(self, nEscCode, count, inData, outData)

Escape Code call to Windows

escapePoint(self, prtEscape)

generic escape which returns a point

flushOutput(self)

Escape FLUSHOUTPUT procedure

getColorTable(self, colorTableIndex)

Escape GETCOLORTABLE procedure. Returns a long integer value corresponding to the RGB color value for the given table entry or nil if unsuccessful

getContext(self)

Return the printer display context for self.

getPhysPageSize(self)

Printer Escape GETPHYSPAGESIZE

getPrinterParms(self)

Get the printer parameters from the WIN.INI file

getPrintingOffset(self)

Printer Escape GETPRINTINGOFFSET

getScalingFactor(self)

Printer Escape GETSCALINGFACTOR

lineTo(self, thePoint)

Draw a line from the current position up to, but not including, self, using the printer handle to a display context. Reset the current printer position to be the receiver Point.

moveTo(self, thePoint)

Change the current printer position to be the receiver point using the printer handle to a device context.

newFrame(self)

Printer Escape NEWFRAME

nextBand(self)

Printer Escape NEXTBAND

queryEscSupport(self, escNum)

Printer Escape QUERYESCSUPPORT

restoreDC(self, nSavedDC)

restore the Device Context.

saveDC(self)

save the DC

setDocName(self, aName)

set the document name to the string

startDoc(self)

Printer Escape STARTDOC

SchedWindow

Source file: SCHEDWIN.CLS

Inherits from: Object WindowsObject Window TextWindow EditWindow HyperWindow TodoWindow TaskWindow

Inherited by: (no descendants)

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
isMain (From class Window)
caption (From class Window)
locRect (From class Window)
buttonDn (From class Window)
hMenu (From class Window)
paintStruct (From class Window)
caretVis (From class TextWindow)
chStr (From class TextWindow)
textMetrics (From class TextWindow)
xMax (From class TextWindow)
yPos (From class TextWindow)
xPos (From class TextWindow)
tmHeight (From class TextWindow)
tmWidth (From class TextWindow)
workText (From class TextWindow)
vCol (From class EditWindow)
aCol (From class EditWindow)
dragDir (From class EditWindow)
endLine (From class EditWindow)
endChar (From class EditWindow)
startLine (From class EditWindow)
startChar (From class EditWindow)
dirty (From class EditWindow)
leftChar (From class EditWindow)
topLine (From class EditWindow)
dragLine (From class EditWindow)
oldX (From class EditWindow)
pOrigin (From class EditWindow)
dragDC (From class EditWindow)
aMenu (From class HyperWindow)
fileStamp (From class HyperWindow)
path (From class HyperWindow)

```
file   (From class HyperWindow)
outputDelimiter    (From class HyperWindow)
inputDelimiter     (From class HyperWindow)
gettingFocus (From class HyperWindow)
hLinkMenu    (From class HyperWindow)
hyperLinks   (From class HyperWindow)
times  (From class TodoWindow)
hWMSize      (From class TodoWindow)
hWindowMenu  (From class TodoWindow)
todo   (From class TodoWindow)
schedule     (From class TaskWindow)
```

Class methods: (none)

Object methods:

addItem(self)

A user may not add an Item to this type of window, This method intercepts the addItem messages and does nothing.

deleteItem(self)

Items may not be deleted from the scheduling window, this method will ignore any messages to the contrary.

help(self)

Open the default help window for this window class.

invSelTxt(self)

Invert the selected text if in view. Assumes a valid dragDC. Check for selected text as a condition to invert.

loadMenu(self, menuName)

Load the menu resource if possible and obtain a handle to a menu to place in hMenu (if menuName not nil).

modifyItem(self)

Modify the selected item. Display a dialog box with any editable fields and replace the item with the modifications if the user choses accept.

paint(self, hdc)

Redraw the workText from topLine down, and from leftChar rightwards, preserve xPos and yPos. If window has focus, show selected text.

readLinks(self)

Replace the contents of the current window with the contents of its associated data file. The Schedule window does not have an associated file, this message is ignored.

setCurPos(self, aPnt)

Set cursor position (xPos, yPos) according to the specified point.

shouldClose(self)

Return true if ok for window to close. The default response is true. Descendant classes can redefine this method if they need to check with user before closing window.

writeLinks(self)

There is not associated data file with a the schedule window. Rather the data is put back into the appropriate task and subtask files.

Schedule

Source file: SCHEDULE.CLS

Inherits from: Object

Inherited by: (no descendants)

This class contains all the methods in the system that pertain to scheduling. Scheduling is done one event at a time and it is assumed that there is no interaction between events. This scheduler performs both the forward and backward passes of cpm.

Instance variables:

currentTask
eventName
tasks
subtaskDict
subtasks

Class methods:

new(self)

Create a schedule instance

Object methods:

calcDates(self,aTask)

find the duration and Due Date of a Task by looking at subtasks

calcSucc(self)

This method calculates the successor list of a the schedule by calculating a closure on the subtasks. (ie any task is the successor of all its predicessors).

collectEvent(self,event)

This method reads in all tasks and subtasks associated with an event in preparation for scheduling calculations.

collectTask(self,task)

This method reads in all subtasks associated with a task in preparation for scheduling calculations. It is usually called by collectEvent.

compressDue(self,asubtask, newDate)

This method changes the due date of a subtask to newDate unless new date is after the current due date of the subtask. The new start date for the subtask is calculated and applied as the new due date to all predicessor subtasks. NOTE: this method assumes the schedule is being compress, it will ignore any message to move a due date forward in time. newDate is a new ending date

compressEvent(self,newDate)

This method sets the due date of an event backward in time. It then sends a message to every subtask associated with the event and forces them to end by the event end date. SEE: compressDue

decompressDue(self,asubtask, newDate)

This method accepts a subtask and a new date for the ending for some one of its predicessors. If the ending date sent is later than the starting date of the subtask the subtask's starting and ending date are adjusted and the new ending date is sent to all the successors of the subtask. newDate is a new ending date for the predicessor task

decompressEvent(self,days)

This method performs moves the start date of an event forward in time. This new start date is then sent to every task in the event to adjust the schedule.

findSlack(self)

This method reviews every subtask in the schedule and calculates the amount of time actually allowed to perform each subtask.

init(self)

Initialize instance variables for a schedule

writeEvent(self,event)

This method rewrites the event file and all the task files with the revised schedule data.

writeKW(self, kwids)

This method appends the todo schedule information in the current schedule to each KW's todo list. Usually, all todo lists would be destroyed, and then each event in the system would be loaded on a time and this method would be called after each one. This would create a complete todo list.

writeTask(self,task)

This method rewrites the schedule information to the specified task file.

Step

Source file: STEP.CLS

Inherits from: Object

Inherited by: (no descendants)

The methods in this class manipulate step objects. A step consists of a title, and a possibly empty collection of attachments, and a possibly empty collection of doits.

Instance variables:

doits collection of doits
attachments collection of attacments
title step title

Class methods:

new(self)

comment

Object methods:

addAttachment(self, attachment)

This method adds the attachment described by the string attachment (in external file format) to the list of attachments associated with the step.

addDoit(self, attachment)

This method adds the doit described by the string attachment (in external file format) to the list of doits associated with the step.

attachments(self)

This method returns the list of attachments associated with the step.

doits(self)

This method returns the list of doits associated with the step.

init(self)

Set instance varibles for a step.

openLink(self,fdir)

This method is called when a step is double clicked on in a step window. It activates a window containing the first item in the list of associated attachments.

printOn(self,aStrm)

This method prints the step in human readable form (external file format)

setTitle(self, aStr)

This method sets the title of a step (the line which appears in a step window).

textColor(self)

This returns the color a step should be displayed in when shown in a step window. This is for compatibility with subtask objects, step always use the default color.

title(self)

This method returns the step title (the line which is displayed in a step window).

windowPrint(self)

This method returns the text that should be displayed in a step window.

StepCollection

Source file: STEPCOLL.CLS

Inherits from: Object Collection IndexedCollection Array OrderedCollection

Inherited by: (no descendants)

This class contains the methods used for manipulating a collection of Steps.

Instance variables:

lastElement (From class OrderedCollection)
firstElement (From class OrderedCollection)

Class methods: (none)

Object methods:

asText(self)

This method returns the step collections as text suitable for printing in the step window.

convertText(self, aTextCol)

This method does nothing (allows this message to be sent to all hypertext type collections)

StepWindow

Source file: STEPWIND.CLS

Inherits from: Object WindowsObject Window TextWindow EditWindow HyperWindow

Inherited by: (no descendants)

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
isMain (From class Window)
caption (From class Window)
locRect (From class Window)
buttonDn (From class Window)
hMenu (From class Window)
paintStruct (From class Window)
caretVis (From class TextWindow)
chStr (From class TextWindow)
textMetrics (From class TextWindow)
xMax (From class TextWindow)
yPos (From class TextWindow)
xPos (From class TextWindow)
tmHeight (From class TextWindow)
tmWidth (From class TextWindow)
workText (From class TextWindow)
vCol (From class EditWindow)
aCol (From class EditWindow)
dragDir (From class EditWindow)
endLine (From class EditWindow)
endChar (From class EditWindow)
startLine (From class EditWindow)
startChar (From class EditWindow)
dirty (From class EditWindow)
leftChar (From class EditWindow)
topLine (From class EditWindow)
dragLine (From class EditWindow)
oldX (From class EditWindow)
pOrigin (From class EditWindow)
dragDC (From class EditWindow)
aMenu (From class HyperWindow)
fileStamp (From class HyperWindow)
path (From class HyperWindow)

```
file   (From class HyperWindow)
outputDelimiter     (From class HyperWindow)
inputDelimiter      (From class HyperWindow)
gettingFocus (From class HyperWindow)
hLinkMenu     (From class HyperWindow)
hyperLinks    (From class HyperWindow)
time
atchMenuSize
hAtchMenu
doitMenuSize
hDoitMenu
```

Class methods: (none)

Object methods:

WM_INITMENUPOPUP(self,wp, lp)

This method is called when a menu item is selected by the user which has a popup menu associated with it. It is called before the menu is displayed. On this screen the only one is the FixIt menu. This method enables or grays the choices available to the user.

addItem(self)

Add an item to this window. A dialog box is presented to the user, if accept is chosen from the dialog box then the item is added below the spot where the selection bar was when the user chose the Add menu choice.

addLinkMenu(self)

This method add any extra menu choices to the base menu bar. Specifically it adds the Attachments and DoIt choices.

convert(self)

This method converts the step data from text format as read in from then step file into the internal step collection.

endDrag(self , wp, lp)

This method changes the line the user has just single clicked on to reverse video (i.e. show the selector bar). It deselects any previously selected item. StartLine will contain the index number to the selected item. Note: the bar is four characters longer than the line length to allow for the step number.

fileName(self)

This method returns the defualt file name for a file associated with the window.

gotFocus(self, hWndNew)

Prepare this window for user input. Also record current time for use in calculating how long the user was working on the information in this window.

help(self)

Open the default help window for this window class.

loadMenu(self, menuName)

Load the menu resource if possible and obtain a handle to a menu to place in hMenu (if menuName not nil).

losingFocus(self, hWndNew)

When losing focus, de-select text visually, and then hide and destroy the caret.

modifyItem(self)

Modify the selected item. Display a dialog box with any editable fields and replace the item with the modifications if the user choses accept.

openAtch(self,num)

This method opens an attachment link. Num contains the index to the link associated with the current item.

openDoit(self,num)

Open a DoIt. num contains the index to the DoIt associated with the current item. If the DoIt is marked executable, call program to run it. Otherwise, Open the doit as an attachment.

paint(self, hdc)

Redraw the workText from topLine down, and from leftChar rightwards, preserve xPos and yPos. If window has focus, show selected text.

setFile(self, fName, apath)

Associate the default file name and default file path with this window.

setTrack(self,file,hPrevWnd)

A step windows is the highest level in the tree of windows we track time for. Its name is the one used for all of its decendants when tracking time spent in a window. Therefor it does not need to handle this message so it ignores it.

updateMenu(self)

This method is called when a user selects a new item in the window. It updates the attachments and DoIt submenues and enables or grays the DoIt and attachments menu selections depending on whether they have submenues or not.

SubTask

Source file: SUBTASK.CLS

Inherits from: Object HyperLink

Inherited by: Task

class comment

Instance variables:

type (From class HyperLink)
file (From class HyperLink)
title (From class HyperLink)
private
impacted node impacted
moved number days node moved
rgbcolor special color to use
longdate due date as long
finished task complete
status project status
days days to complete

Class methods: (none)

Object methods:

<(self,aLink)

This method is used to add a subtask to a sorted collection, it sorts by fininished by date

complete(self)

returns number of seconds spent working on subtask.

due(self)

This method returns the due date of a subtask in military format.

finished(self)

This task returns true if the subtask is finished.

longDate(self)

This method returns the due date as a long int.

openLink(self,fdir)

This method opens a step window containing the step file associated with the subtask.

printOn(self,aStrm)

This method prints the object on the stream specified.

setColor(self)

This method calculates the color the subtask should be displayed in when it is being printed on the screen.

setDue(self, aStr)

This method sets the due date associated with a subtask to the date contained in the string aStr.

setLDue(self,along)

This method sets the date associated with a subtask to the date (specified as a long int) passed in along.

setLink(self ,aStr)

Translate a string containing printed format into internal format

setStatus(self,aStr)

This method sets the amount of time worked on a task to the int value passed in aStr.

textColor(self)

This routine returns the color that a subtask should be displayed in on the screen.

toggleFinished(self)

This method marks a finished subtask unfinished and vice-versa.

windowPrint(self)

this method returns as a string the information a user would see in a window

SubtaskCollection

Source file: SUBTASKC.CLS

Inherits from: Object Collection IndexedCollection Array
OrderedCollection SortedCollection LinkCollection

Inherited by: TaskCollection

This class contains the methods used in manipulating a collection of subtask links

Instance variables:

lastElement (From class OrderedCollection)
firstElement (From class OrderedCollection)
compareBlock (From class SortedCollection)

Class methods: (none)

Object methods:

addString(self,aStr)

This method adds a string which contains the external format of a subtask link (the format stored in a file) into the collection in the internal format. The link is inserted in the normal sorting order, by finished by date.

insertLinkStr(self, aStr, idx)

This method adds a string which contains the external format of a subtask link (the format stored in a file) into the collection in the internal format. The link is inserted at the positon specified by idx.

Task

Source file: TASK.CLS

Inherits from: Object HyperLink SubTask

Inherited by: (no descendants)

class comment

Instance variables:

type (From class HyperLink)
file (From class HyperLink)
title (From class HyperLink)
private (From class SubTask)
impacted (From class SubTask)
moved (From class SubTask)
rgbcolor (From class SubTask)
longdate (From class SubTask)
finished (From class SubTask)
status (From class SubTask)
days (From class SubTask)
assigned
task
after
before
duration

Class methods:

new(self)

create a new task

Object methods:

addAft(self,str)

This routine adds the task name passed in str to the list of successors associated with the subtask.

addb4(self,str)

This routine adds the task name passed in str to the list of predicessors associated with the subtask.

after(self)

This method returns the collection of succesors associated with this task.

before(self)

This method returns the collection of predicessors associated with this task.

dur(self)

This method returns the estimated duration associated with a task.

init(self)

This method initializes the instance variables associated with a task object.

kwid(self)

This method returns the Knowledge Worker ID of the KW assigned to the task.

printOn(self,aStrm)

This method prints the object on the stream specified.

setDur(self,val)

This method associates an estimated duration (string) with a task.

setKwid(self,val)

This method associates a KW with a task. Val is the KWid of the KW assigned to a task.

setLink(self ,aStr)

Translate a string containing printed format into internal format

setTaskID(self,val)

This method associates the task id with a task.

taskid(self)

This method returns the task id associated with a task.

windowPrint(self)

this method returns as a string the information a user would see in a window

TaskCollection

Source file: TASKCOLL.CLS

Inherits from: Object Collection IndexedCollection Array
OrderedCollection SortedCollection LinkCollection
SubtaskCollection

Inherited by: (no descendants)

this class contains the methods used to manipulate a collection of tasks

Instance variables:

lastElement (From class OrderedCollection)
firstElement (From class OrderedCollection)
compareBlock (From class SortedCollection)
taskList

Class methods: (none)

Object methods:

addString(self,aStr)

This method adds a string which contains the external format of a task link (the format stored in a file) into the collection in the internal format. The link is inserted in the normal sorting order, by finished by date.

calcSucc(self)

This method calculates the successor links within a collection of tasks which have VALID PREDICESSOR links.

find(self,link)

This method returns the task which is linked to the file specified by link.

insertLinkStr(self, aStr, idx)

This method adds a string which contains the external format of a task link (the format stored in a file) into the collection in the internal format. The link is inserted at the positon specified by idx.

TaskLWindow

Source file: TASKLWIN.CLS

Inherits from: Object WindowsObject Window TextWindow EditWindow HyperWindow TodoWindow EventLWindow

Inherited by: (no descendants)

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
isMain (From class Window)
caption (From class Window)
locRect (From class Window)
buttonDn (From class Window)
hMenu (From class Window)
paintStruct (From class Window)
caretVis (From class TextWindow)
chStr (From class TextWindow)
textMetrics (From class TextWindow)
xMax (From class TextWindow)
yPos (From class TextWindow)
xPos (From class TextWindow)
tmHeight (From class TextWindow)
tmWidth (From class TextWindow)
workText (From class TextWindow)
vCol (From class EditWindow)
aCol (From class EditWindow)
dragDir (From class EditWindow)
endLine (From class EditWindow)
endChar (From class EditWindow)
startLine (From class EditWindow)
startChar (From class EditWindow)
dirty (From class EditWindow)
leftChar (From class EditWindow)
topLine (From class EditWindow)
dragLine (From class EditWindow)
oldX (From class EditWindow)
pOrigin (From class EditWindow)
dragDC (From class EditWindow)
aMenu (From class HyperWindow)
fileStamp (From class HyperWindow)
path (From class HyperWindow)

```
file   (From class HyperWindow)
outputDelimiter    (From class HyperWindow)
inputDelimiter     (From class HyperWindow)
gettingFocus (From class HyperWindow)
hLinkMenu    (From class HyperWindow)
hyperLinks   (From class HyperWindow)
times  (From class TodoWindow)
hWMSize      (From class TodoWindow)
hWindowMenu  (From class TodoWindow)
todo   (From class TodoWindow)
updates      (From class EventLWindow)
hasChild     (From class EventLWindow)
myEvent
```

Class methods: (none)

Object methods:

fileName(self)

This method returns the defualt file name for a file associated with the window.

help(self)

Open the default help window for this window class.

linkTo(self, aLink)

This method opens a window for the link passed in aLink.

loadMenu(self, menuName)

Load the menu resource if possible and obtain a handle to a menu to place in hMenu (if menuName not nil).

scheduleItem(self)

This method starts the dynamic scheduler on the event associated with the this task.

setEvent(self,anEvent)

This method associates the name of the event a task is in with the task. This information is used when starting the dynamic sheduler.

TaskWindow

Source file: TASKWIND.CLS

Inherits from: Object WindowsObject Window TextWindow EditWindow HyperWindow TodoWindow

Inherited by: SchedWindow

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
isMain (From class Window)
caption (From class Window)
locRect (From class Window)
buttonDn (From class Window)
hMenu (From class Window)
paintStruct (From class Window)
caretVis (From class TextWindow)
chStr (From class TextWindow)
textMetrics (From class TextWindow)
xMax (From class TextWindow)
yPos (From class TextWindow)
xPos (From class TextWindow)
tmHeight (From class TextWindow)
tmWidth (From class TextWindow)
workText (From class TextWindow)
vCol (From class EditWindow)
aCol (From class EditWindow)
dragDir (From class EditWindow)
endLine (From class EditWindow)
endChar (From class EditWindow)
startLine (From class EditWindow)
startChar (From class EditWindow)
dirty (From class EditWindow)
leftChar (From class EditWindow)
topLine (From class EditWindow)
dragLine (From class EditWindow)
oldX (From class EditWindow)
pOrigin (From class EditWindow)
dragDC (From class EditWindow)
aMenu (From class HyperWindow)
fileStamp (From class HyperWindow)
path (From class HyperWindow)

```
file   (From class HyperWindow)
outputDelimiter    (From class HyperWindow)
inputDelimiter     (From class HyperWindow)
gettingFocus (From class HyperWindow)
hLinkMenu    (From class HyperWindow)
hyperLinks   (From class HyperWindow)
times  (From class TodoWindow)
hWMSize      (From class TodoWindow)
hWindowMenu  (From class TodoWindow)
todo   (From class TodoWindow)
schedule
```

Class methods:

view(self,parent, title,fPath,schedule)

create a new Task window showing the subtasks in a "schedule". it is the parents job to collect and write out the schedule data.

Object methods:

addItem(self)

Add an item to this window. A dialog box is presented to the user, if accept is chosen from the dialog box then the item is added in the the window according to sorting order. (by finished by date)

fileName(self)

This method returns the defualt file name for a file associated with the window.

help(self)

Open the default help window for this window class.

initWorkText(self)

Initialize the workText instance variable, add one zero-length string.

modifyItem(self)

Modify the selected item. Display a dialog box with any editable fields and replace the item with the modifications if the user choses accept.

readLinks(self)

Replace the contents of the current window with the contents of its associated data file. The file is read one subtask at a time and then inserted into workText. A subtask can be a multiline entry in the data

file. The first line is the title, followed by any number of attachment links. The first character of each line describes what it contains. It is blank for a title, - for a predicessor, and + for a successor. Finally, a convert message is sent to workText to handle any class specific initialization that may be necessary.

shouldClose(self)

Return true if ok for window to close. The default response is true. Descendant classes can redefine this method if they need to check with user before closing window.

showSelf(self,subtasks)

This method is used to open a task window without an associated data file.

TextPrinter

Source file: TEXTPRIN.CLS

Inherits from: Object Printer

Inherited by: (no descendants)

General Text Print Class to output text to a printer. Mark Solinski, The Whitewater Group

Instance variables:

port (From class Printer)
driver (From class Printer)
device (From class Printer)
docName (From class Printer)
hPrintDC (From class Printer)
chStr
textMetrics
yPos Current y position
yMax Max Rows
xPos Current x position
xMax Max columns

Class methods: (none)

Object methods:

drawChar(self, aChar)

Draw a character in the window at current position. Go the next line if character is a CR.

drawString(self, aStr)

Draw a string to the printer.

eol(self)

##finish(self)

closes down the print document

height(self)

return the tmHeight

home(self)

Reset the x and y position variables

init(self)

Initialize the text printer class

newPage(self)

Home the cursor position and perform a new frame

print(self, object)

print the object string out to printer

printLine(self, object)

print the object string out to printer, with CR_LF at end

start(self, docName)

create a new document with the passed name

testForNewPage(self)

##textPixelWidth(self, aStr)

returns the width in pixels of the passed string in the current font

width(self)

return the tmWidth

x(self)

Translate xPos and return current x coordinate in pixels.

y(self)

Translate yPos and return current y coordinate in pixels.

TodoWindow

Source file: TODOWIND.CLS

Inherits from: Object WindowsObject Window TextWindow EditWindow HyperWindow

Inherited by: EventLWindow TaskLWindow KWTodoWindow TaskWindow SchedWindow

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
isMain (From class Window)
caption (From class Window)
locRect (From class Window)
buttonDn (From class Window)
hMenu (From class Window)
paintStruct (From class Window)
caretVis (From class TextWindow)
chStr (From class TextWindow)
textMetrics (From class TextWindow)
xMax (From class TextWindow)
yPos (From class TextWindow)
xPos (From class TextWindow)
tmHeight (From class TextWindow)
tmWidth (From class TextWindow)
workText (From class TextWindow)
vCol (From class EditWindow)
aCol (From class EditWindow)
dragDir (From class EditWindow)
endLine (From class EditWindow)
endChar (From class EditWindow)
startLine (From class EditWindow)
startChar (From class EditWindow)
dirty (From class EditWindow)
leftChar (From class EditWindow)
topLine (From class EditWindow)
dragLine (From class EditWindow)
oldX (From class EditWindow)
pOrigin (From class EditWindow)
dragDC (From class EditWindow)
aMenu (From class HyperWindow)
fileStamp (From class HyperWindow)

path (From class HyperWindow)
file (From class HyperWindow)
outputDelimiter (From class HyperWindow)
inputDelimiter (From class HyperWindow)
gettingFocus (From class HyperWindow)
hLinkMenu (From class HyperWindow)
hyperLinks (From class HyperWindow)
times a dictionary of user times
hWMSize a dictionary of user times
hWindowMenu a dictionary of user times
todo collection of subtasks

Class methods: (none)

Object methods:

WM_INITMENUPOPUP(self,wp, lp)

This method is called when a menu item is selected by the user which has a popup menu associated with it. It is called before the menu is displayed. On this screen the are two, one is the FixIt menu, the other is the Windows menu. This method enables or grays the choices available to the user on the FixIt menu, and updates the list of currently open KW windows.

addItem(self)

Add an item to this window. A dialog box is presented to the user, if accept is chosen from the dialog box then the item is added in the the window according to sorting order. (by finished by date)

addLinkMenu(self)

This method add any extra menu choices to the base menu bar. Specifically it adds the Windows choice.

convert(self)

sets startline to item closest to current date

defaultFdir(self)

This method overides the method in HyperWindow and returns the private data path or the KW data path depending on whether the currently selected subtask is a private or scheduler subtask.

fileName(self)

This method returns the defualt file name for a file associated with the window.

finished(self)

This method handled the finished menu choice. It marks the item finished or unfinished as appropriate, then move the item to its new position in the window.

help(self)

Open the default help window for this window class.

initWorkText(self)

Initialize the workText instance variable, add one zero-length string.

loadMenu(self, menuName)

Load the menu resource if possible and obtain a handle to a menu to place in hMenu (if menuName not nil).

logTime(self,atask,atime)

This method adds the time spent in any step or attachment windows for a subtask to the total time spent working on the subtask.

modifyItem(self)

Modify the selected item. Display a dialog box with any editable fields and replace the item with the modifications if the user choses accept.

openDoit(self,num)

This really brings to the top the specified window

readLinks(self)

Replace the contents of the current window with the contents of its associated data file. The file is read one line at a time and then inserted into workText. Finally, a convert message is sent to workText to handle any class specific initialization that may be necessary. Additionally, Todo windows keep track of the file timestamp and will reread the file if it has been changed outside the system.

setFile(self, fName, apath)

Associate the default file name and default file path with this window.

ViewTodo

Source file: VIEWTODO.CLS

Inherits from: Object WindowsObject Dialog FileDialog EditTodo

Inherited by: (no descendants)

class comment

Instance variables:

cRect (From class WindowsObject)
parent (From class WindowsObject)
defProc (From class WindowsObject)
hWnd (From class WindowsObject)
editFocus (From class FileDialog)
pathSpec (From class FileDialog)
fileSpec (From class FileDialog)
loadFile (From class FileDialog)
startDir (From class FileDialog)
task (From class EditTodo)

Class methods: (none)

Object methods:

initDialog(self, wP, lP)

This method initializes an edit todo dialog box, but disables editting on all fields the KW is not allowed to change.

KNOWLEDGE WORKER SYSTEM PROTOTYPE:

LESSONS LEARNED

Prepared Under Contract
DACA88-88-D-0020/0008

Task 3-3

Submitted to:

Department of the Army
CONSTRUCTION ENGINEERING RESEARCH LABORATORY
Champaign, Illinois

Prepared January 12, 1990 by:
GEORGIA INSTITUTE OF TECHNOLOGY
Atlanta, Georgia

Introduction

The purpose of this report is to review the development of the Knowledge Worker System (KWS) prototype. The strengths and weakness of the approach taken will be discussed. Based upon lessons learned in the development of this prototype, ideas for future Knowledge Worker System development will be presented.

The Knowledge Worker System Prototype was developed by the Georgia Institute of Technology (GA Tech) during calendar year 1989. This work was performed under contract for the United States Army, Construction Engineering Research Laboratory (USACERL). GA Tech worked closely with USACERL's Management and Knowledge Integration (MKI) team and with John Sheehey, U.S. Army Corps of Engineers, Military Programs Directorate, Programming and Execution Support Office (CEMP-P), Chief, to develop specifications for the prototype.

The KWS prototype was installed on four workstations in CEMP-P. Fifteen users - members of the KWS User Group that represent seven U.S. Army Corps of Engineers (USACE) and Department of the Army (DA) offices - attended KWS prototype demonstration/training sessions that were offered in CEMP-P in Oct 89.

The KWS prototype was initially intended to be a tool that could be used to rapidly experiment with different ideas for the functionality of a production version of the KWS system. Two key areas were intended for experimentation: the user interface and the scheduling system. The intention of this development strategy was to subsequently use the results of this experimentation to design a production version of KWS. As the project progressed, however, pressure for a working version of KWS mandated changes to the initial goals so that the prototype was turned into a functional pilot system.

Strengths and Weaknesses

The design of the KWS prototype was influenced by three requirements: 1) Maximum compatibility with existing computer hardware and software; 2) Provision of an easy-to-use, enticing user interface; and 3) Completion of the software in time for presentation to the targeted users at a Joint Application Design (JAD) workshop for the KWS User Group in Nov 89.

The enormous world-wide investment by the Army in IBM- compatible personal computer equipment essentially dictated the use of IBM-compatible personal computers as the KWS hardware platform. Developers did, however, exclude from consideration any personal computer which did not have a 386-class machine, as a minimum.

The requirement that KWS users have 386-class machines was a result of

the decision to use Microsoft Windows/386. Windows/386 was selected because of its ability to run existing DOS programs without modification. This capability is very important because otherwise, users' existing software would have to be replaced.

Basing the KWS prototype on a 386-class provides another advantage, namely, KWS software can run on an Zenith 248 PC (a 286-class machine) by upgrading the Z-248 with the Aox Corporation's Z-Master board. The Z-Master board replaces the 286 processor on the Zenith computer with a 386 processor. This relatively inexpensive conversion allows users to run the KWS prototype software on the Zenith machines, which is an important consideration in light of the Army's purchase of over 250,000 Zenith 248 PCs under the Department of Defense Standard Desktop Computer Contract. USACERL provided GA Tech with a Zenith 248 that was converted into a 386 and successfully used for KWS development and testing.

One disadvantage of the requirement for a 386-class machine is that it precludes the use of the KWS prototype on a large number of AT-type PCs. The office where the prototype was installed (CEMP-P) uses Compaq/20e machines, which are in the 386 class and therefore, within the context of a prototype system, the specification of a 386-class machine was not problematic. In other offices, however, the 386 requirement may prove to be limiting; this problem may be offset by the use of converted Zenith 248s, as described above.

The KWS prototype was developed with the Actor programming language to run as a Microsoft Windows application. These choices enabled GA Tech developers to implement the prototype in a relatively short time period and to provide a consistent user interface that will be compatible with the SAA standard. The compatibility with SAA will guarantee compatibility with future IBM operating systems and help minimize the impact of moving KWS to future operating systems for the PC, such as OS/2.

The selection of Actor for developing a prototype was a good one for several reasons. In fact, this choice was nearly mandated; without the software development tools provided with Actor, it probably would have been impossible to implement the KWS software within the required time frame. MS Windows programs are complex and the interactive development environment provided by Actor greatly expedited the development process. In addition to an interactive development environment, Actor provides a large number of predefined functions for writing Windows-based programs. Finally, since Actor is an object-oriented language, it reduced the amount of code that had to be written to create the prototype. As well, the resulting KWS software is flexible and therefore changes to prototype functionality can be effected more easily than if KWS had been generated with a procedural language.

Actor does have some limitations which may preclude its use when writing the production version of KWS. Actor is an interpreted language; while this fact speeds up the development process by allowing the effects of changes to be seen quickly, it slows down the execution speed of the

program. Secondly, Actor is a proprietary language and may limit the systems which to which KWS can be ported.

A language such as C++ seems a better choice for implementing the production version of KWS. C++ offers the advantages of object-oriented programming that Actor provides. However, C++ is a compiled language which allows the resulting programs to run faster. Additionally, C++ is available under a large number of operating system.

The selection of the DOS operating system resulted in several problems. The major problem was caused by the limitation DOS places on the size of executable programs. Actor programs tend to be very large. This fact, combined with the memory requirement for MS Windows itself, produces a severe limitation on future extensions to KWS within this environment.

Because of the memory limitations of DOS, the production version should be targeted to another platform such as OS/2. The OS/2 platform has two major advantages. First, this choice will expand the base of supported hardware to include AT-class machines as well as 386- class machines. Secondly, migration to OS/2 will relieve memory limitations associated with using DOS. OS/2 was not selected for prototype development because of the lack of application software. OS/2 is an attractive option for the production version because of the increase in the number of applications that either can run under OS/2 now or will be available in the immediate future.

Finally, time constraints reduced the degree of attention given to the scheduling aspects of KWS. A relevant factor that constantly hindered the development effort was the lack of a centralized database system. KWS needs to be used by people in widely-separated sites. The nature of data access required to perform scheduling requires that all time-related data be available to both the scheduling program and to individual users. These considerations imply the need for a centralized data base or a tightly-coupled system of database servers.

The Programming, Administration, and eXecution (PAX) system will need to be enhanced in order to accommodate the scheduling function. The most important outstanding task to be completed before finalizing the design of the production version of KWS will be to determine and implement the PAX enhancements. It will be important to integrate personal computers which are not directly connected into any local area network into this solution.

Conclusion

In general design assumptions proved to be good ones. The software and machine choices, while somewhat limiting in power, enabled system developers to provide the KWS prototype to users quickly and at minimal cost. Time constraints necessarily limited the scope of the effort. With the help of the users and the prototype the Joint Application Design (JAD) workshop produced a good set of user interface

requirements. One major area of study remains, i.e., the database required for the scheduling portion of the KWS software.

The original intention in USACERL's development strategy was that the prototype would be completely rewritten before being provided to users as a finished product. However, user demand for the product was so strong that prototype design was modified in order to add enough functionality that the prototype, as delivered, qualifies as a working system, albeit limited in scope. This decision was made because it enabled users to get involved in the design process early. Early user involvement will logically lead to the development of a final product that meets user requirements. This consideration outweighs the extra development effort involved in adding functionality to an interim product.

Recommendations

Based on our experience in implementing the prototype, we recommend the following changes for the production version.

(1) There absolutely must be an underlying database system. This system must provide, as a minimum, the ability to handle transaction requests from user's personal computers. It must allow stand-alone personal computers and, personal computers connected to local area networks, to make these transaction requests.

(2) The DOS operating system will eventually have to be abandoned. The production version of KWS should not be designed to run on DOS. OS/2 looks like a very good choice.

(3) C++ is recommended as the software to implement the production version of KWS. One caveat: if Actor runs under the targeted operating system, the rewrite of KWS prototype code could be deferred until either speed considerations or the desire to port KWS to various operating systems forced a rewrite of the code.